Capítulo 7

RAMIFICACIÓN Y PODA

7.1 INTRODUCCIÓN

Este método de diseño de algoritmos es en realidad una variante del diseño Vuelta Atrás estudiado en el capítulo anterior. Sin embargo, su particular importancia y extenso uso hace que nosotros le dediquemos un capítulo aparte.

Esta técnica de diseño, cuyo nombre en castellano proviene del término inglés *Branch and Bound*, se aplica normalmente para resolver problemas de optimización. Ramificación y Poda, al igual que el diseño Vuelta Atrás, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.

Una característica que le hace diferente al diseño anterior es la posibilidad de generar nodos siguiendo distintas estrategias. Recordemos que el diseño Vuelta Atrás realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones. El diseño Ramificación y Poda en su versión más sencilla puede seguir un recorrido en anchura (estrategia LIFO) o en profundidad (estrategia FIFO), o utilizando el cálculo de funciones de coste para seleccionar el nodo que en principio parece más prometedor (estrategia de mínimo coste o LC).

Además de estas estrategias, la técnica de Ramificación y Poda utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

Definimos *nodo vivo* del árbol a un nodo con posibilidades de ser ramificado, es decir, que no ha sido podado. Para determinar en cada momento que nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos en alguna estructura que podamos recorrer. Emplearemos una pila para almacenar los nodos que se han generado pero no han sido examinados en una búsqueda en profundidad (LIFO). Las búsquedas en amplitud utilizan una cola (FIFO) para almacenar los nodos vivos de tal manera que van explorando nodos en el mismo orden en que son creados. La estrategia de mínimo coste (LC) utiliza una *función de coste* para decidir en cada momento qué nodo debe explorarse, con la esperanza de alcanzar lo más rápidamente posible una

solución más económica que la mejor encontrada hasta el momento. Utilizaremos en este caso una estructura de montículo (o cola de prioridades) para almacenar los nodos ordenados por su coste.

Básicamente, en un algoritmo de Ramificación y Poda se realizan tres etapas. La primera de ellas, denominada de *Selección*, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo. En la segunda etapa, la *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior. Por último se realiza la tercera etapa, la *Poda*, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección. El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos.

Para cada nodo del árbol dispondremos de una función de coste que nos estime el valor óptimo de la solución si continuáramos por ese camino. De esta manera, si la cota que se obtiene para un nod, que por su propia construcción deberá ser mejor que la solución real (o a lo sumo, igual que ella), es peor que una solución ya obtenida por otra rama, podemos podar esa rama pues no es interesante seguir por ella. Evidentemente no podremos realizar ninguna poda hasta que hayamos encontrado alguna solución. Por supuesto, las funciones de coste han de ser crecientes respecto a la profundidad del árbol, es decir, si h es una función de coste entonces $h(n) \leq h(n')$ para todo n' nodo descendiente de n.

En consecuencia, y a la vista de todo esto, podemos afirmar que lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. La dificultad está en encontrar una buena función de coste para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso. Si es demasiado simple probablemente pocas ramas puedan ser excluidas. Dependiendo de cómo ajustemos la función de coste mejor algoritmo se deriva.

Inicialmente, y antes de proceder a la poda de nodos, tendremos que disponer del coste de la mejor solución encontrada hasta el momento que permite excluir de futuras expansiones cualquier solución parcial con un coste mayor. Como muchas veces no se desea esperar a encontrar la primera solución para empezar a podar, un buen recurso para los problemas de optimización es tomar como mejor solución inicial la obtenida con un algoritmo ávido, que como vimos no encuentra siempre la solución óptima, pero sí una cercana a la óptima.

Por último, sólo comentar una ventaja adicional que poseen estos algoritmos: la posibilidad de ejecutarlos en paralelo. Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda. El disponer de algoritmos paralelizables (y estos algoritmos, así como los de Divide y Vencerás lo son) es muy importante en muchas aplicaciones en las que es necesario abordar los problemas de forma paralela para resolverlos en tiempos razonables, debido a su complejidad intrínseca.

Sin embargo, todo tiene un precio, sus requerimientos de memoria son mayores que los de los algoritmos Vuelta Atrás. Ya no se puede disponer de una estructura global en donde ir construyendo la solución, puesto que el proceso de construcción deja de ser tan "ordenado" como antes. Ahora se necesita que cada nodo sea autónomo, en el sentido que ha de contener toda la información necesaria para realizar los procesos de bifurcación y poda, y para reconstruir la solución encontrada hasta ese momento.

7.2 CONSIDERACIONES DE IMPLEMENTACIÓN

Uno de las dificultades que suele plantear la técnica de Ramificación y Poda es la implementación de los algoritmos que se obtienen. Para subsanar este problema, en esta sección presentamos una estructura general de tales algoritmos, basada en tres módulos (en el sentido de Modula-2) principales:

1. De un lado dispondremos del módulo que contiene el esquema de funcionamiento general de este tipo de algoritmos.
2. Por otro se encuentra el módulo que maneja la estructura de datos en donde se almacenan los nodos que se van generando, y que puede tratarse de una pila, una cola o un montículo (según se siga una estrategia LIFO, FIFO o LC).
3. Finalmente, necesitamos un módulo que describa e implemente las estructuras de datos que conforman los nodos.

El primero de los tres módulos no se modifica a lo largo del capítulo, pues es válido para todos los algoritmos que siguen la técnica de Ramificación y Poda, y lo presentamos a continuación:

```
MODULE Esquema;
FROM IO IMPORT WrStr, WrCard, WrLn;
FROM Estruc IMPORT Estructura,Crear,Anadir,Extraer,EsVacia,
                   Tamano,Destruir;
FROM Nodos IMPORT nodo,NodoInicial,MAXHIJOS,Expandir,EsAceptable,
              EsSolucion,h,Eliminar,NoHaySolucion,Imprimir,PonerCota;
VAR  numgenerados,        (* numero total de nodos generados *)
     numanalizados,       (* numero total de nodos analizados *)
     numpodados:CARDINAL; (* numero total de nodos podados *)

PROCEDURE RyP_una():nodo; (* encuentra la primera solucion *)
   VAR E:Estructura; (* estructura para almacenar los nodos *)
       n:nodo; (* nodo vivo en curso *)
       hijos:ARRAY [1..MAXHIJOS] OF nodo; (* hijos de un nodo *)
       numhijos,i,j:CARDINAL;
BEGIN
   E:=Crear();   (* inicializamos las estructuras *)
   n:=NodoInicial(); Anadir(E,n,h(n)); (*h es la funcion de coste*)
   WHILE NOT EsVacia(E) DO
      n:=Extraer(E); INC(numanalizados);
      numhijos:=Expandir(n,hijos); INC(numgenerados,numhijos);
      Eliminar(n);
```

```
      FOR i:=1 TO numhijos DO
         IF EsAceptable(hijos[i]) THEN
            IF EsSolucion(hijos[i]) THEN (* Eureka! *)
               FOR j:=1 TO numhijos DO (*eliminamos resto de hijos*)
                  IF i<>j THEN Eliminar(hijos[j]) END;
               END;
               Destruir(E);
               RETURN hijos[i]  (* devolvemos la solucion *)
            ELSE
               Anadir(E,hijos[i],h(hijos[i]))
            END;
         ELSE
            Eliminar(hijos[i]); INC(numpodados)
         END;
      END;
   END;
   Destruir(E);
   RETURN NoHaySolucion();
END RyP_una;

(* programa principal del esquema *)
   VAR n:nodo;
BEGIN
   numgenerados:=0;  numanalizados:=0; numpodados:=0;
   n:=RyP_una();
   WrStr("Nodos Generados:  "); WrCard(numgenerados,4); WrLn();
   WrStr("Nodos Analizados: "); WrCard(numanalizados,4); WrLn();
   WrStr("Nodos Podados:    "); WrCard(numpodados,4); WrLn();
END Esquema.
```

Como podemos ver, además de encontrar una solución, el programa calcula tres datos, el número de nodos generados, el número de nodos analizados, y el número de nodos podados, los cuales permiten analizar el algoritmo y poder comparar distintas estrategias y funciones LC.

a) El primero de ellos (*numgenerados*) nos da información sobre el trabajo que ha tenido que realizar el algoritmo hasta encontrar la solución. Mientras más pequeño sea este valor, menos parte del árbol de expansión habrá tenido que construir, y por tanto más rápido será el proceso.

b) El segundo valor (*numanalizados*) nos indica el número de nodos que el algoritmo ha tenido que analizar, para lo cual es necesario extraerlos de la estructura y comprobar si han de ser podados y, si no, expandirlos. Éste es el valor más importante de los tres, pues indica el número de nodos del árbol de expansión que se recorren efectivamente. En consecuencia, es deseable que este valor sea pequeño.

c) Por último, el número de nodos podados nos da una indicación de la efectividad de la función de poda y las restricciones impuestas al problema. Mientras mayor sea este valor, más trabajo ahorramos al algoritmo.

Disponer de esta forma fácil y modular de cambiar las estrategias de selección de los nodos vivos (mediante el módulo "Estruc") junto con los valores de estos tres parámetros nos permitirá analizar el algoritmo de Ramificación y Poda de una forma sencilla, cómoda y eficaz, y en consecuencia escoger la mejor de las estrategias para un problema dado.

Si lo que deseamos es encontrar no sólo una solución al problema sino todas, observamos que es posible conseguirlo con una pequeña variación del esquema anterior:

```
PROCEDURE RyP_todas(VAR todas:ARRAY OF nodo):CARDINAL;
(* encuentra todas las soluciones del problema y
   devuelve el numero de soluciones que hay *)
VAR E:Estructura; (* estructura para almacenar los nodos *)
    n:nodo; (* nodo vivo en curso *)
    hijos:ARRAY [1..MAXHIJOS] OF nodo; (* hijos de un nodo*)
    numhijos,i,j,numsol:CARDINAL;
BEGIN
   E:=Crear();
   n:=NodoInicial();
   Anadir(E,n,h(n));
   numsol:=0;
   WHILE NOT EsVacia(E) DO (* analiza todo el arbol *)
      n:=Extraer(E); INC(numanalizados);
      numhijos:=Expandir(n,hijos); INC(numgenerados,numhijos);
      Eliminar(n);
      FOR i:=1 TO numhijos DO
         IF EsAceptable(hijos[i]) THEN
            IF EsSolucion(hijos[i]) THEN (* Eureka! *)
               todas[numsol]:=hijos[i]; INC(numsol)
            ELSE
               Anadir(E,hijos[i],h(hijos[i]))
            END;
         ELSE
            Eliminar(hijos[i]); INC(numpodados)
         END;
      END;
   END;
   Destruir(E);
   RETURN numsol;
END RyP_todas;
```

También vamos a considerar una tercera versión del algoritmo para cuando necesitemos encontrar la mejor de entre todas las soluciones de un problema:

```
PROCEDURE RyP_lamejor():nodo;
VAR E:Estructura; (* estructura para almacenar los nodos *)
    n,solucion:nodo;
    hijos:ARRAY [1..MAXHIJOS] OF nodo; (* hijos de un nodo*)
    numhijos,i,j,valor,valor_solucion:CARDINAL;
BEGIN
   E:=Crear(); n:=NodoInicial(); Anadir(E,n,h(n));
   solucion:=NoHaySolucion(); valor_solucion:=MAX(CARDINAL);
   PonerCota(valor_solucion);
   WHILE NOT EsVacia(E) DO
      n:=Extraer(E); INC(numanalizados);
      numhijos:=Expandir(n,hijos); INC(numgenerados,numhijos);
      Eliminar(n);
      FOR i:=1 TO numhijos DO
         IF EsAceptable(hijos[i]) THEN
            IF EsSolucion(hijos[i]) THEN (* Eureka! *)
               valor:=Valor(hijos[i]);
               IF valor<valor_solucion THEN
                  Eliminar(solucion); solucion:=hijos[i];
                  valor_solucion:=valor; PonerCota(valor);
               END;
            ELSE
               Anadir(E,hijos[i],h(hijos[i]))
            END;
         ELSE
            Eliminar(hijos[i]); INC(numpodados)
         END;
      END;
   END;
   Destruir(E);
   RETURN solucion;
END RyP_lamejor;
```

Una vez disponemos del esquema de este tipo de algoritmos, vamos a definir el interfaz de los tipos abstractos de datos que representan los nodos y la estructura de datos para almacenarlos. En primer lugar, el módulo Nodos ha de implementar los siguientes procedimientos y funciones:

```
DEFINITION MODULE Nodos;
CONST MAXHIJOS = ...; (* numero maximo de hijos de un nodo *)
TYPE nodo;
PROCEDURE NodoInicial():nodo; (* raiz del arbol *)
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
```

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
PROCEDURE h(n:nodo):CARDINAL;
PROCEDURE PonerCota(c:CARDINAL);
PROCEDURE Valor(n:nodo):CARDINAL;
PROCEDURE Eliminar(VAR n:nodo);
PROCEDURE NoHaySolucion():nodo;
PROCEDURE Imprimir(n:nodo);
END Nodos.
```

- De ellas, *NodoInicial* es la que devuelve el nodo que constituye la raíz del árbol de expansión para el problema. A partir de este nodo se origina el árbol, expandiendo progesivamente los nodos mediante la siguiente función.
- *Expandir* es la función que construye los nodos hijos de un nodo dado, y devuelve el número de hijos que ha generado. Esta función es la que realiza el proceso de ramificación del algoritmo.
- La función *EsAceptable* es la que realiza la poda, y dado un nodo vivo decide si seguir analizándolo o bien rechazarlo.
- *EsSolucion* es una función que decide cuándo un nodo es una hoja del árbol, esto es, una posible solución al problema original. Obsérvese que tal solución no ha de ser necesariamente la mejor, sino una cualquiera de ellas.
- Por su parte, la función *h* es la que implementa la función de coste de la estrategia LC, y que se utiliza como prioridad en la estructura de montículo.
- La función *Valor* devuelve el valor asociado a un nodo, y se utiliza para comparar soluciones con la cota superior encontrada hasta el momento.
- La función *PonerCota* permite establecer la cota superior del problema. Usualmente la función que realiza la poda utiliza este dato para podar aquellos nodos cuyo valor (calculado mediante la función *Valor*) sea superior a la cota ya obtenida de una solución. El código de esta función va a ser común para los ejemplos desarrollados en este capítulo, y por lo tanto lo incluimos aquí:

  ```
  PROCEDURE PonerCota(c:CARDINAL);
  BEGIN
     cota:=c;
  END PonerCota;
  ```

  siendo *cota* una variable global (aunque privada) del módulo "Nodos" que se utiliza para almacenar la cota inferior del problema alcanzada hasta ese momento por alguna solución. Nótese que hablamos de cota inferior puesto que el esquema presentado permite resolver problemas de minimización. En el ejemplo de la mochila presentado en este capítulo se discute la solución de los problemas de maximización.
- La función *NoHaySolucion* es la que devuelve un nodo con valor especial, necesario para indicar que no se encuentra solución al problema.
- Por último, las funciones *Eliminar* e *Imprimir* son las que destruyen e imprimen un nodo, respectivamente.

En cuanto al tipo abstracto de datos que representa la estructura donde se almacenan los nodos, su interfaz es el siguiente:

```
DEFINITION MODULE Estruc;
   FROM Nodos IMPORT nodo;
   TYPE Estructura;
PROCEDURE Crear():Estructura;
PROCEDURE Anadir(VAR h:Estructura;n:nodo;prioridad:CARDINAL);
PROCEDURE Extraer(VAR h:Estructura):nodo;
PROCEDURE EsVacia(h:Estructura):BOOLEAN;
PROCEDURE Tamano(h:Estructura):CARDINAL;
PROCEDURE Destruir(VAR h:Estructura);
END Estruc.
```

Hemos llamado a este tipo abstracto *Estructura* porque puede corresponder a una cola, una pila o un montículo invertido (en la raíz del árbol se encuentra el menor elemento puesto que tratamos con problemas de minimización) dependiendo de la estrategia que queramos implementar en nuestro algoritmo de Ramificación y Poda (FIFO, LIFO o LC). No consideramos necesario incluir su implementación, pues ni aporta nada nuevo a la técnica ni presenta ninguna dificultad especial.

Utilizando este esquema conseguimos reducir la programación de los algoritmos de Ramificación y Poda a la implementación de las funciones que conforman los nodos, lo que reduce notablemente la dificultad de implementación de este tipo de algoritmos. Por consiguiente, para la resolución de los problemas planteados en este capítulo será suficiente dar una implementación del módulo “Nodos” de cada uno de ellos.

7.3 EL PUZZLE (n^2–1)

Este juego es una generalización del Puzzle-15 ideado por Sam Loyd en 1878. Disponemos de un tablero con n^2 casillas y de n^2–1 piezas numeradas del uno al n^2–1. Dada una ordenación inicial de todas las piezas en el tablero, queda sólo una casilla vacía, a la que denominaremos “hueco”. Nuestro objetivo es transformar, mediante movimientos legales de la fichas, dicha disposición inicial de las piezas en una disposición final ordenada, en donde en la casilla $[i,j]$ se encuentra la pieza numerada $(i–1)*n + j$ y en la casilla $[n,n]$ se encuentra el hueco.

Los únicos movimientos permitidos son los de las piezas adyacentes (horizontal y verticalmente) al hueco, que pueden ocuparlo; al hacerlo, dejan el hueco en la posición en donde se encontraba la pieza antes del movimiento.

Otra forma de abordar el problema es considerar que lo que se mueve es el hueco, pudiendo hacerlo hacia arriba, abajo, izquierda o derecha (siempre sin salirse del tablero). Al moverse, su casilla es ocupada por la pieza que ocupaba la casilla a donde se ha “movido” el hueco. Por ejemplo, para el caso $n = 3$ se muestra a continuación una disposición inicial junto con la disposición final:

Disposición Inicial Disposición Final

1	5	2
4	3	
7	8	6

1	2	3
4	5	6
7	8	

Es posible resolver el problema mediante Ramificación y Poda utilizando dos funciones de coste diferentes:

a) La primera calcula el número de piezas que están en una posición distinta de la que les corresponde en la disposición final.

b) La segunda se basa en la suma de las distancias de Manhattan desde la posición de cada pieza a su posición en la disposición final. La distancia de Manhattan entre dos puntos del plano de coordenadas (x_1,y_1) y (x_2,y_2) viene dada por la expresión $|x_1 - x_2| + |y_1 - y_2|$.

Se pide resolver este problema utilizando ambas funciones de coste, y comparar los resultados que se obtienen para ambas funciones.

Solución (☞)

Para resolver este problema es necesario en primer lugar construir su árbol de expansión, y para ello hemos de plantearlo como una secuencia de decisiones, una en cada nivel del árbol.

Por tanto, partiendo de una disposición del tablero, consideraremos como posibles decisiones a tomar los cuatro movimientos del hueco (arriba, abajo, izquierda y derecha) siempre que éstos sean válidos, es decir, siempre que no caigan fuera del tablero.

Así por ejemplo, partiendo de la disposición inicial mostrada en el enunciado del problema tenemos tres opciones válidas:

1	5	
4	3	2
7	8	6

1	5	2
4	3	6
7	8	

1	5	2
4		3
7	8	6

A partir de esta idea vamos a construir el módulo de implementación asociado a los nodos que es, según hemos comentado en la introducción de este capítulo, lo único que necesitamos para resolver el problema.

Una primera aproximación a la solución del problema consiste en definir cada uno de los nodos como un tablero, es decir:

```
CONST dim = ...; (* dimension del puzzle *)
TYPE puzzle = ARRAY [1..dim],[1..dim] OF CARDINAL;
TYPE nodo = POINTER TO puzzle;
```

Sin embargo, esto no va a ser suficiente puesto que no disponemos de "historia" sobre los movimientos ya realizados, lo cual nos llevaría posiblemente a ciclos en donde repetiríamos indefinidamente movimientos de forma circular.

Aparte de esta razón, también hemos de recordar que los nodos utilizados en la técnica de Ramificación y Poda han de ser autónomos, es decir, han de contener toda la información necesaria para recuperar a partir de ellos la solución construida hasta el momento. En nuestro caso la solución ha de estar compuesta por una sucesión de tableros que muestran la serie de movimientos a realizar para llegar a la disposición final.

Por tanto, la definición de nodos que vamos a utilizar es:

```
CONST dim = ...; (* dimension del puzzle *)
TYPE puzzle = ARRAY [1..dim],[1..dim] OF CARDINAL;
TYPE nodo = POINTER TO RECORD p:puzzle; sig:nodo; END;
```

Otra posibilidad sería la de utilizar una lista global con todos aquellos tableros que ya han sido considerados, y que fuera utilizada durante el proceso de bifurcación de todos los nodos para comprobar que no se generan duplicados. De esta forma también se eliminarían los ciclos. La diferencia de propuesta es que esa lista sería global a todos los nodos, mientras que en la primera cada nodo tiene la lista de los nodos que él ha generado. En este problema haremos uso de la primera de las opciones.

Una vez disponemos de la representación de los valores del tipo abstracto de datos, implementaremos sus operaciones. En primer lugar, la función *NodoInicial* habrá de contener la disposición inicial del tablero:

```
PROCEDURE NodoInicial():nodo;
   VAR x:nodo;
BEGIN
   NEW(x);
   x^.p[1,1]:=1; x^.p[1,2]:=5; x^.p[1,3]:=2;
   x^.p[2,1]:=4; x^.p[2,2]:=3; x^.p[2,3]:=0;
   x^.p[3,1]:=7; x^.p[3,2]:=8; x^.p[3,3]:=6;
   x^.sig:=NIL;
   RETURN x;
END NodoInicial;
```

La estrategia de ramificación está a cargo de la función *Expandir*:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR i,j,nhijos:CARDINAL;
       p:nodo;
BEGIN
   BuscaHueco(n,i,j); (* primero buscamos el "hueco" *)
   nhijos:=0;
   (* y ahora vemos a donde lo podemos "mover" *)
   IF i<dim THEN (* abajo *)
```

```
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i+1,j]; p^.p[i+1,j]:=0;
      hijos[nhijos-1]:=p;
   END;
   IF j<dim THEN (* derecha *)
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i,j+1]; p^.p[i,j+1]:=0;
      hijos[nhijos-1]:=p;
   END;
   IF (i-1)>0 THEN (* arriba *)
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i-1,j]; p^.p[i-1,j]:=0;
      hijos[nhijos-1]:=p;
   END;
   IF (j-1)>0 THEN (* izquierda *)
      INC(nhijos);
      Copiar(n,p);
      p^.p[i,j]:=p^.p[i,j-1]; p^.p[i,j-1]:=0;
      hijos[nhijos-1]:=p;
   END;
   RETURN nhijos;
END Expandir;
```

Una de las primeras dudas que nos asaltan tras implementar esta función es si el orden en el que se bifurque va a influir en la eficiencia del algoritmo, tal como sucedía en algunos problemas de Vuelta Atrás. Realmente, el orden de ramificación sí es importante cuando el árbol de expansión se recorre siguiendo una estrategia "ciega" (FIFO o LIFO). Sin embargo, puesto que en este problema vamos a utilizar una estrategia LC, el orden en el que se generen los nodos (y se inserten en la estructura) no va a tener una influencia de peso en el comportamiento final del algoritmo.

Esto también lo hemos probado de forma experimental, pues una vez obtenidos los resultados finales decidimos cambiar este orden, y generar nodos moviendo el hueco en el sentido inverso a las agujas del reloj y comenzando por arriba (a priori es la peor manera, pues el hueco ha de tratar de ir hacia la posición [*n*,*n*]). Los resultados obtenidos de esta forma no presentan ningún cambio sustancial respecto a los anteriores, lo que corrobora el hecho de que en este caso el orden de generación de los hijos no es influyente. En cualquier caso, esta afirmación es cierta para este problema pero no tiene por qué ser válida para cualquier otro; obsérvese que en este caso el número de hijos que genera cada nodo es pequeño (a lo más cuatro). Para problemas en los que el número de hijos que expande cada nodo es grande, sí que puede tener influencia el orden de generación de los mismos.

Volviendo a la función *Expandir*, su implementación hace uso de varios procedimientos auxiliares, que presentamos a continuación:

```
PROCEDURE BuscaHueco(n:nodo;VAR i,j:CARDINAL);
(* busca el hueco en un tablero *)
   VAR a,b:CARDINAL;
BEGIN
   FOR a:=1 TO dim DO
      FOR b:=1 TO dim DO
         IF n^.p[a,b]=0 THEN
            i:=a; j:=b; RETURN
         END
      END
   END
END BuscaHueco;
```

También necesita una función para copiar un nodo y añadirle el tablero que va a contener el siguiente movimiento:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL; nuevo:nodo;
BEGIN
   NEW(n2);
   Duplicar(n1,nuevo);
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         n2^.p[i,j]:=n1^.p[i,j]
      END
   END;
   n2^.sig:=nuevo;
END Copiar;
```

Esta función utiliza la que duplica un nodo dado:

```
PROCEDURE Duplicar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL;
BEGIN
   NEW(n2);
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         n2^.p[i,j]:=n1^.p[i,j]
      END
   END;
   n2^.sig:=n1^.sig;
   IF n1^.sig<>NIL THEN Duplicar(n1^.sig,n2^.sig) END
END Duplicar;
```

Continuando con la implementación del módulo "Nodos", también es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuyo último movimiento haga aparecer un ciclo:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
(* mira si ese movimiento ya lo ha hecho antes *)
   VAR aux:nodo;
BEGIN
   aux:=n^.sig;
   WHILE aux<>NIL DO
      IF SonIguales(n,aux) THEN RETURN FALSE END;
      aux:=aux^.sig;
   END;
   RETURN TRUE;
END EsAceptable;
```

A su vez, esta función utiliza otra que permite decidir cuándo dos tableros son iguales:

```
PROCEDURE SonIguales(n1,n2:nodo):BOOLEAN;
VAR i,j:CARDINAL;
BEGIN
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         IF n1^.p[i,j]<>n2^.p[i,j] THEN RETURN FALSE END
      END
   END;
   RETURN TRUE;
END SonIguales;
```

Una función que también es necesario implementar es la que define la función de coste. Para este problema vamos a implementar dos, una para cada una de las estrategias mencionadas en el enunciado. La primera de ellas va a contar el número de piezas que se encuentran fuera de su sitio:

```
PROCEDURE h(n:nodo):CARDINAL;
(* cuenta el numero de piezas fuera de su posicion final *)
   VAR i,j,cuenta:CARDINAL;
BEGIN
   cuenta:=0;
   FOR i:=1 TO dim DO FOR j:=1 TO dim DO
       IF n^.p[i,j]<>((j+(i-1)*dim)MOD(dim*dim)) THEN INC(cuenta) END
   END END;
   RETURN cuenta;
END h;
```

La segunda corresponde a la suma de las distancias de Manhattan de la posición de cada pieza a su casilla final, y que hace uso de una función que calcula el valor absoluto de la diferencia de dos números naturales:

```
PROCEDURE ValAbs(a,b:CARDINAL):CARDINAL;
(* valor absoluto de la diferencia de sus argumentos: |a-b| *)
BEGIN
   IF a>b THEN RETURN a-b ELSE RETURN b-a END;
END ValAbs;

PROCEDURE h2(n:nodo):CARDINAL;
(* calcula la suma de las distancias de Manhattan *)
   VAR i,j,x,y,cuenta:CARDINAL;
BEGIN
   cuenta:=0;
   FOR i:=1 TO dim DO
      FOR j:=1 TO dim DO
         IF n^.p[i,j] = 0 THEN
            x:=dim; y:=dim
         ELSE
            x:=((n^.p[i,j]-1) DIV dim)+1;
            y:=((n^.p[i,j]-1) MOD dim)+1;
         END;
         cuenta:=cuenta+ValAbs(x,i)+ValAbs(y,j);
      END
   END;
   RETURN cuenta;
END h2;
```

También es preciso implementar una función para determinar cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo un tablero coincide con la disposición final del juego y para esto es suficiente comprobar que su función de coste vale cero (cualquiera de las dos funciones vistas):

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN h(n)=0;
END EsSolucion;
```

También es necesario implementar la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución:

```
PROCEDURE NoHaySolucion():nodo;
BEGIN
   RETURN NIL;
END NoHaySolucion;
```

Obsérvese que esto puede ocurrir puesto que no todas las disposiciones iniciales de un puzzle permiten llegar a la disposición final, como por ejemplo ocurre para la siguiente disposición inicial:

1	3	2
4	5	6
7	8	

Por último, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, actuando como destructor del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   IF n<>NIL THEN
      IF n^.sig<>NIL THEN Eliminar(n^.sig) END;
      DISPOSE(n);
   END;
END Eliminar;
```

El procedimiento *Imprimir* no plantea mayores problemas, y su implementación va a depender de lo que el usuario desee consultar sobre un nodo. En cuanto a las funciones *Valor* y *PonerCota*, como nuestro problema consiste en encontrar la primera solución, no tienen relevancia alguna.

Una vez implementado el módulo "Nodos" es el momento de analizar su comportamiento. Para ello haremos uso de los resultados que nos da el programa principal que contiene el esquema, y que mostramos en las siguientes tablas. Cada una de ellas contiene a la izquierda la disposición inicial de partida, y los valores obtenidos utilizando cada una de las dos funciones LC que hemos implementado. Hemos llamado LC_1 a la función de coste que contaba el número de piezas fuera de su sitio, y LC_2 a la otra.

Disposición Inicial

1	5	2
4	3	
7	8	6

Resultados Obtenidos

	LC_1	LC_2
Núm. nodos generados	19	23
Núm. nodos analizados	7	8
Núm. nodos podados	5	6

1	3	5
7		2
8	4	6

	LC_1	LC_2
Núm. nodos generados	38	48
Núm. nodos analizados	13	17
Núm. nodos podados	11	15

4	1	5
7		2
8	3	6

	LC_1	LC_2
Núm. nodos generados	47	194
Núm. nodos analizados	17	71
Núm. nodos podados	15	69

Como puede apreciarse, la primera función de coste se comporta de forma más eficaz que la segunda.

7.4 EL VIAJANTE DE COMERCIO

Analicemos una vez más el problema del viajante de comercio, presentado ya en el capítulo cuatro, y cuyo enunciado reza como sigue. Se conocen las distancias entre un cierto número de ciudades. Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible. Más formalmente, dado un grafo *g* conexo y ponderado, y dado uno de sus vértices v_0, queremos encontrar el ciclo Hamiltoniano de coste mínimo que comienza y termina en v_0.

Solución

El problema del viajante de comercio admite numerosas estrategias de ramificación y poda, y casi cada autor que describe el problema emplea una distinta, o incluso varias. Nosotros utilizaremos la primera de las tres estrategias descritas en [HOR78] para solucionar el problema.

Comenzaremos analizando la construcción del árbol de expansión para el problema. En primer lugar, hemos de plantear la solución como una secuencia de decisiones, una en cada paso o etapa.

Para ello, nuestra solución estará formada por un vector que va a indicar el orden en el que deberán ser visitados los vértices. Cada elemento del vector contendrá un número entre 1 y *N*, siendo *N* el número de vértices del grafo que define el problema. Es preciso indicar aquí que utilizaremos una representación del grafo en donde los vértices están numerados consecutivamente comenzando por 1, y los arcos vienen definidos mediante una matriz de adyacencia, no necesariamente simétrica en este caso, aunque sí de elementos no negativos.

De esta forma, inicialmente el vector solución estará compuesto por un solo elemento, el 1 (que es el vértice origen), y en cada paso *k* tomaremos la decisión de

qué vértice incluimos en el recorrido. Por tanto, los valores que puede en principio tomar el elemento en posición *k* del vector ($1 \le k \le N$) estarán comprendidos entre 1 y *N* pero sin poder repetirse, esto es, no puede haber dos elementos iguales en el vector. Por tanto, cada nodo podrá generar hasta *N–k* hijos. Este mecanismo es el que va construyendo el árbol de expansión para el problema.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de datos que representa los nodos, pues el resto del programa es fijo para estos algoritmos.

Respecto a la información que debe contener cada uno de ellos, hemos de conseguir que cada nodo sea "autónomo", esto es, que cada uno contenga toda la información relevante para poder realizar los procesos de bifurcación, poda y reconstrucción de la solución encontrada hasta ese momento. En consecuencia, al menos ha de contar con el nivel en donde se encuentra y con el vector solución construido hasta el momento. Por otro lado, también debe llevar información para realizar la poda. En este sentido vamos a incluir una *matriz de costes reducida* .

Diremos que una fila (columna) de una matriz está *reducida* si contiene al menos un elemento cero, y el resto de los elementos son no negativos. Una matriz se dice *reducida* si y sólo si todas sus filas y columnas están reducidas. Por ejemplo, dada la matriz de adyacencia:

∞	15	7	4	20
1	∞	16	6	5
8	20	∞	4	10
4	7	14	∞	3
10	35	15	4	∞

podemos calcular su matriz reducida restando respectivamente 4, 1, 4, 3 y 4 a cada fila, y luego 4 y 3 a las columnas 2 y 3, obteniendo la matriz:

∞	7	0	0	16
0	∞	12	5	4
4	12	∞	0	6
1	0	8	∞	0
6	27	8	0	∞

En total hemos restado un valor de 23 (4 + 1 + 4 + 3 + 4 + 4 + 3), que es lo que denominaremos el *coste* de la matriz.

De esta forma, dada una matriz de adyacencia de un grafo ponderado podemos obtener su matriz reducida calculando los mínimos de cada una de las filas y restándoselos a los elementos de esas filas, haciendo después lo mismo con las columnas.

Respecto a la interpretación del *coste*, pensemos que restando una cantidad t a una fila o a una columna decrementamos en esa cantidad el coste de los recorridos del grafo. Por tanto, un camino mínimo lo seguirá siendo tras una operación de sustracción de filas o columnas. En cuanto a la cantidad total sustraída al reducir una matriz, ésta será una cota inferior del coste total de sus recorridos. En consecuencia, para el ejemplo anterior hemos obtenido que 23 es una cota inferior para la solución al problema del viajante.

Esto es justo lo que vamos a utilizar como función de coste LC para podar nodos del árbol de expansión. Así, a cada nodo le vamos a asociar una matriz reducida y un coste acumulado. Para ver cómo trabajamos con ellos, supongamos que M es la matriz reducida asociada al nodo n, y sea n' el hijo de n que se obtiene incluyendo el arco $\{i,j\}$ en el recorrido.

- Si n' es una hoja del árbol, esto es, una posible solución, su coste va a venir dado por el coste que llevaba n acumulado más $M[i,j]+M[j,1]$, que es lo que completa el recorrido. Esta cantidad coincide además con el coste de tal recorrido.
- Por otro lado, si n' no es una hoja, su matriz de costes reducida M' vamos a calcularla a partir de los valores de M como sigue:

  a) En primer lugar, hay que sustituir todos los elementos de la fila i y de la columna j por ∞. Esto elimina el posterior uso de aquellos caminos que parten del vértice i y de los que llegan al vértice j.

  b) En segundo lugar, debemos asignar $M'[j,1]=\infty$, eliminando la posibilidad de acabar el recorrido en el siguiente paso (recordemos que n' no era una hoja).

  c) Reducir entonces la matriz M', y ésta es la matriz que asignamos al nodo n'.

  Como coste para n' vamos a tomar el coste de n más el coste de la reducción de M' más, por supuesto, el valor de $M[i,j]$. Es importante señalar en este punto que la reducción no se realiza teniendo en cuenta los elementos con valor ∞, no obteniéndose coste alguno en aquellas filas o columnas cuyos elementos tomen todos tal valor.

Con todo esto, comenzaremos definiendo el tipo *nodo* que utilizaremos en la implementación del algoritmo de Ramificación y Poda que resuelve el problema. Vamos a utilizar entonces la siguiente estructura de datos:

```
CONST N = ...; (* numero de vertices del grafo *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE mat_ady = ARRAY[1..N],[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                         coste:CARDINAL; (* coste acumulado *)
                         matriz:mat_ady; (* matriz reducida *)
                         k:CARDINAL;     (* nivel *)
                         s:solucion
                       END;
```

Además del vector solución y el nivel, los otros componentes del registro indican el coste acumulado hasta el momento, así como la matriz reducida asociada al nodo.

Necesitaremos también una variable global al módulo para almacenar la cota superior alcanzada por la mejor solución hasta el momento:

```
VAR cota:CARDINAL;
```

Esta variable será inicializada en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de contener el nodo raíz del árbol de expansión:

```
PROCEDURE NodoInicial():nodo;
    VAR n:nodo; i,j:CARDINAL; m:mat_ady;
BEGIN
    (* aqui se introduce la matriz de adyacencia del grafo *)
    FOR i:=1 TO N DO m[i,i]:=MAX(CARDINAL) END;
               m[1,2]:=15; m[1,3]:=7 ; m[1,4]:=4 ; m[1,5]:=20;
    m[2,1]:=1 ;            m[2,3]:=16; m[2,4]:=6 ; m[2,5]:=5 ;
    m[3,1]:=8 ; m[3,2]:=20;            m[3,4]:=4 ; m[3,5]:=10;
    m[4,1]:=4 ; m[4,2]:=7 ; m[4,3]:=14;            m[4,5]:=3 ;
    m[5,1]:=10; m[5,2]:=35; m[5,3]:=15; m[5,4]:=4 ;
    (* ahora, generamos el primer nodo *)
    NEW(n);
    FOR i:=2 TO N DO n^.s[i]:=0 END;
    n^.matriz:=m;
    n^.coste:=Reducir(n^.matriz);
    n^.s[1]:=1; (* incluimos el primer vertice *)
    n^.k:=1;
    RETURN n;
END NodoInicial;
```

Como podemos observar, se introduce ya en la solución el vértice origen, y la matriz que se asocia a este nodo es la reducida de la original. El procedimiento que se encarga de la reducción es el siguiente:

```
PROCEDURE Reducir(VAR m:mat_ady):CARDINAL;
(* devuelve el coste total reducido a la matriz *)
    VAR i,j,coste,minimo:CARDINAL;
BEGIN
    coste:=0;
    FOR i:=1 TO N DO (* primero por filas *)
```

```
        minimo:=CosteFil(m,i);
        IF (minimo>0) AND (minimo<MAX(CARDINAL)) THEN
            QuitarFil(m,i,minimo); INC(coste,minimo)
        END
    END;
    FOR j:=1 TO N DO (* despues por columnas *)
        minimo:=CosteCol(m,j);
        IF (minimo>0) AND (minimo<MAX(CARDINAL)) THEN
            QuitarCol(m,j,minimo); INC(coste,minimo)
        END
    END;
    RETURN coste;
END Reducir;
```

Para lograr su objetivo, se basa en los procedimientos que calculan el mínimo de una fila y se lo restan a los elementos de tal fila, y los análogos para las columnas:

```
PROCEDURE CosteFil(m:mat_ady;i:CARDINAL):CARDINAL;
    VAR j,c:CARDINAL;
BEGIN
    c:=m[i,1];
    FOR j:=2 TO N DO
        IF m[i,j]<c THEN c:=m[i,j] END;
    END;
    RETURN c
END CosteFil;

PROCEDURE CosteCol(m:mat_ady;j:CARDINAL):CARDINAL;
    VAR i,c:CARDINAL;
BEGIN
    c:=m[1,j];
    FOR i:=2 TO N DO
        IF m[i,j]<c THEN c:=m[i,j] END;
    END;
    RETURN c
END CosteCol;

PROCEDURE QuitarFil(VAR m:mat_ady;i:CARDINAL;minimo:CARDINAL);
    VAR j:CARDINAL;
BEGIN
    FOR j:=1 TO N DO
        IF m[i,j]<MAX(CARDINAL) THEN m[i,j]:=m[i,j]-minimo END;
    END;
END QuitarFil;
```

```
PROCEDURE QuitarCol(VAR m:mat_ady;j:CARDINAL;minimo:CARDINAL);
    VAR i:CARDINAL;
BEGIN
    FOR i:=1 TO N DO
        IF m[i,j]<MAX(CARDINAL) THEN m[i,j]:=m[i,j]-minimo END;
    END;
END QuitarCol;
```

Por otro lado, la estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar, como hemos dicho antes, a lo sumo *N–k* hijos, que son los correspondientes a los vértices aún no incluidos en el recorrido.

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR nk,i,j,l,coste,nhijos:CARDINAL;
    p:nodo;
BEGIN
    nhijos:=0;
    nk:=n^.k+1;
    i:=n^.s[nk-1];
    IF nk>N THEN  (* caso especial *)
        RETURN nhijos
    END;
    FOR j:=1 TO N DO
        IF NoEsta(n^.s,nk-1,j) THEN
            INC(nhijos);
            Copiar(n,p);
            p^.s[nk]:=j;
            IF nk=N THEN (* recorrido completo *)
                INC(p^.coste,n^.matriz[i,j]+n^.matriz[j,1])
            ELSE
              FOR l:=1 TO N DO
                  p^.matriz[i,l]:=MAX(CARDINAL);
                  p^.matriz[l,j]:=MAX(CARDINAL);
              END;
              p^.matriz[j,1]:=MAX(CARDINAL);
              INC(p^.coste,Reducir(p^.matriz)+n^.matriz[i,j]);
            END;
            INC(p^.k);
            hijos[nhijos-1]:=p;
        END
    END;
    RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
    VAR i,j:CARDINAL;
BEGIN
    NEW(n2);
    n2^.s:=n1^.s; n2^.matriz:=n1^.matriz;
    n2^.coste:=n1^.coste; n2^.k:=n1^.k;
END Copiar;
```

Y también de otra función para determinar si un vértice del grafo está ya incluido o no en el recorrido:

```
PROCEDURE NoEsta(s:solucion;k,j:CARDINAL):BOOLEAN;
    VAR i:CARDINAL;
BEGIN
    FOR i:=1 TO k DO
        IF s[i]=j THEN RETURN FALSE END
    END;
    RETURN TRUE;
END NoEsta;
```

Es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuya penalización hasta el momento supere la alcanzada por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN n^.coste;
END Valor;
```

que devuelve el coste acumulado hasta el momento. Esto tiene sentido pues el objetivo es encontrar la solución de menor coste.

Veamos ahora la función de coste para los nodos. Como nos piden encontrar la solución de menor coste, este valor es el más adecuado para tal función:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
   RETURN n^.coste;
END h;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de acomodar hasta el *N*-ésimo vértice:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca si el grafo es conexo, pues siempre existe al menos una solución, que es la que conecta a todos los vértices entre sí.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

Para los valores iniciales del ejemplo, el algoritmo encuentra un recorrido óptimo de coste 29, que es el representado por el vector solución [1, 3, 5, 4, 2], obteniéndose los siguientes valores de exploración del árbol de expansión:

Núm. nodos generados	26
Núm. nodos analizados	12
Núm. nodos podados	14

Nos podemos plantear también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro modelo de programación, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola. Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente.

	LC	LIFO	FIFO
Núm. nodos generados	26	36	64
Núm. nodos analizados	12	18	41
Núm. nodos podados	14	13	18

Como era de esperar por la función de coste definida para el problema, el mejor caso se obtiene cuando la estrategia de búsqueda es LC. Como veremos en otros ejemplos esto no es siempre así, pues para ciertos problemas no existen funciones de coste que permitan agilizar de forma notable la búsqueda por el árbol. De hecho, la búsqueda de buenas funciones de coste para reducir la exploración del árbol de expansión de un problema es una de las partes más delicadas e importantes de su resolución, sobre todo en aquellos casos en donde el árbol sea, por su tamaño, intratable.

7.5 EL LABERINTO

Este problema fue presentado en el apartado 6.6 del capítulo anterior, y consiste en determinar el camino de salida de un laberinto, representado por una matriz que indica las casillas transitables.

Solución (☺)

Cara a resolver este problema utilizando Ramificación y Poda, podemos definir una función LC basándonos en la distancia de Manhattan del punto en donde nos encontramos actualmente hasta la casilla de salida, es decir, el número mínimo estimado de movimientos para alcanzar la salida.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de datos que representa los nodos, pues el resto del programa es fijo.

Para ello, comenzaremos definiendo el tipo *nodo*. En primer lugar, deberá ser capaz no sólo de representar el laberinto y en dónde nos encontramos en el momento dado, sino que además deberá contener información sobre el recorrido realizado hasta tal punto. Utilizaremos por tanto las siguientes estructuras:

```
CONST dim = ...;     (* dimension del laberinto *)
TYPE laberinto = ARRAY[1..dim],[1..dim] OF CARDINAL;
TYPE nodo =   POINTER TO RECORD
                 x,y:CARDINAL;
                 l:laberinto
              END;
```

Las coordenadas *x* e *y* indican la casilla en donde nos encontramos, y los valores que vamos a almacenar en la matriz que define el laberinto indican el estado en el que se encuentra cada casilla, pudiendo ser:

a) 0 si la casilla no ha sido visitada,
b) *MAX(CARDINAL)* si la casilla no es transitable, o
c) un valor entre 1 y *dim*dim* que indica el orden en el que la casilla ha sido visitada.

De esta forma conseguimos que cada nodo sea "autónomo", esto es, que cada uno contenga toda la información relevante para poder realizar los procesos de bifurcación, la poda y la reconstrucción de la solución encontrada hasta ese momento. Necesitaremos además una variable global al módulo para almacenar la cota superior alcanzada por la mejor solución hasta el momento:

```
VAR cota:CARDINAL; (* num. movimientos de la mejor solucion *)
```

Esta variable será inicializada en el cuerpo del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
END Nodos.
```

Respecto a las funciones de este módulo, en primer lugar la función *NodoInicial* habrá de contener la disposición inicial del laberinto:

```
CONST MURO = MAX(CARDINAL);

PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i,j:CARDINAL;
BEGIN
   NEW(n);
   (* rellenamos a cero el laberinto *)
   FOR i:=1 TO dim DO FOR j:=1 TO dim DO n^.l[i,j]:=0 END END;
   (* situamos la casilla inicial *)
   n^.x:=1; n^.y:=1;
   n^.l[1,1]:=1;
   (* y ahora ponemos los bloques que forman los muros *)
   n^.l[1,5]:=MURO; n^.l[2,3]:=MURO; n^.l[3,2]:=MURO;
   n^.l[3,3]:=MURO; n^.l[3,5]:=MURO; n^.l[4,3]:=MURO;
   n^.l[4,5]:=MURO; n^.l[5,1]:=MURO; n^.l[5,3]:=MURO;
   n^.l[6,5]:=MURO;
   RETURN n;
END NodoInicial;
```

Siendo *MURO* una constante con el valor *MAX(CARDINAL)*. El laberinto representado por esa disposición es el siguiente:

1				X	
		X			

	X	X		X	
		X		X	
X		X			
				X	

La estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar hasta cuatro hijos, que son los correspondientes a los posibles movimientos que podemos realizar desde una casilla dada (arriba, abajo, izquierda, derecha). Esta función sólo generará aquellos movimientos que sean válidos, esto es, que no se salgan del laberinto, no muevan sobre un muro, o bien sobre una casilla previamente visitada:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR i,j,nhijos:CARDINAL; p:nodo;
BEGIN
    nhijos:=0;
    i:=n^.x;
    j:=n^.y;
    (* y ahora vemos a donde lo podemos "mover" *)
    IF ((i-1)>0) AND (n^.l[i-1,j]=0) THEN (* arriba *)
        INC(nhijos);
        Copiar(n,p);
        p^.l[i-1,j]:=p^.l[i,j]+1;
        DEC(p^.x);
        hijos[nhijos-1]:=p;
    END;
    IF ((j-1)>0) AND (n^.l[i,j-1]=0) THEN (* izquierda *)
        INC(nhijos);
        Copiar(n,p);
        p^.l[i,j-1]:=p^.l[i,j]+1;
        DEC(p^.y);
        hijos[nhijos-1]:=p;
    END;
    IF (i<dim) AND (n^.l[i+1,j]=0) THEN (* abajo *)
        INC(nhijos);
        Copiar(n,p);
        p^.l[i+1,j]:=p^.l[i,j]+1;
        INC(p^.x);
        hijos[nhijos-1]:=p;
    END;

    IF (j<dim) AND (n^.l[i,j+1]=0) THEN (* derecha *)
        INC(nhijos);
        Copiar(n,p);
```

```
        p^.l[i,j+1]:=p^.l[i,j]+1;
        INC(p^.y);
        hijos[nhijos-1]:=p;
     END;
     RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL;
BEGIN
   NEW(n2);
   FOR i:=1 TO dim DO FOR j:=1 TO dim DO
      n2^.l[i,j]:=n1^.l[i,j]
   END END;
   n2^.x:=n1^.x;
   n2^.y:=n1^.y;
END Copiar;
```

Una de las primeras dudas que nos asaltan tras implementar la función *Expandir* es si el orden en el que se bifurque va a influir en la eficiencia del algoritmo, tal como sucedía en algunos problemas de Vuelta Atrás. Realmente, el orden de ramificación sí es importante cuando el árbol de expansión se recorre siguiendo una estrategia "ciega" (FIFO o LIFO). Sin embargo, puesto que en este problema vamos a utilizar una estrategia LC, el orden en el que se generen los nodos (y se inserten en la estructura) no va a tener una influencia de peso en el comportamiento final del algoritmo.

Esto también lo hemos probado de forma experimental, y los resultados obtenidos muestran que el comportamiento del algoritmo no varía sustancialmente cuando se altera el orden en que se generan los nodos. En cualquier caso, esta afirmación es cierta para este problema pero no tiene por qué ser válida para cualquier otro: obsérvese que en este caso el número de hijos que genera cada nodo es pequeño (a lo más cuatro). Para problemas en los que el número de hijos que expande cada nodo es grande sí que puede tener influencia el orden de generación de los mismos.

Por otro lado, también es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuyo recorrido hasta el momento supere el número de pasos alcanzado por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
```

```
BEGIN
   RETURN n^.l[n^.x,n^.y];
END Valor;
```

que devuelve el número de pasos dados hasta el momento en el recorrido.

Veamos ahora la función de coste para los nodos. Tal como nos piden en el enunciado del problema, ésta corresponde a la distancia de Manhattan desde la posición en la que nos encontramos a la casilla final:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
   RETURN (dim-n^.x)+(dim-n^.y);
END h;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos llegado a la casilla final, y para esto es suficiente comprobar que su función de coste vale cero:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN h(n)=0
END EsSolucion;
```

También es necesario implementar la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución:

```
PROCEDURE NoHaySolucion():nodo;
BEGIN
   RETURN NIL;
END NoHaySolucion;
```

Obsérvese que esto puede ocurrir para algunos laberintos, si es que los muros "rodean" completamente la salida. Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema apropiada según deseemos encontrar una solución, todas, o la mejor.

Para el valor inicial que damos en este ejemplo, los valores obtenidos por el programa son los que a continuación mostramos.

- En el caso de buscar solamente una solución, la primera que se encuentra consta de 13 movimientos y es la siguiente:

1				X	
2		X			
3	X	X		X	
4	5	X		X	
X	6	X	10	11	12
	7	8	9	X	13

Y los valores que se obtienen son:

Núm. nodos generados	17
Núm. nodos analizados	12
Núm. nodos podados	0

- En el caso de buscar la mejor solución, ésta consta de 11 movimientos y es la siguiente:

1	2	3	4	X	
		X	5		
	X	X	6	X	
		X	7	X	
X		X	8	9	10
				X	11

Y los valores que se obtienen en este caso son:

Núm. nodos generados	75
Núm. nodos analizados	62
Núm. nodos podados	11

- En el caso de buscar todas las soluciones, se consigue hallar 8 soluciones distintas, de longitudes 13, 19, 11, 11, 13, 13, 15 y 21 respectivamente, y los valores que se obtienen en este caso son:

Núm. nodos generados	166
Núm. nodos analizados	159
Núm. nodos podados	0

Nos podemos plantear también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro modelo de programación, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola. Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente.

- En el caso de la estrategia LIFO los resultados que se obtienen no varían demasiado respecto a los conseguidos siguiendo nuestra estrategia LC:

	Primera	Mejor	Todas
Núm. nodos generados	15	69	166
Núm. nodos analizados	10	58	159
Núm. nodos podados	0	10	0

Como era de esperar, el valor de la columna "Todas" es igual, puesto que el árbol se rastrea completamente. Los valores de las otras dos columnas son similares a los obtenidos para la estrategia LC; el hecho que sean un poco mejores depende sólo del ejemplo concreto. Para otros ejemplos los valores que se obtienen siguiendo esta estrategia son peores (p.e. para aquellos laberintos con pocas casillas no transitables).

- En el caso de la estrategia FIFO los resultados son los siguientes:

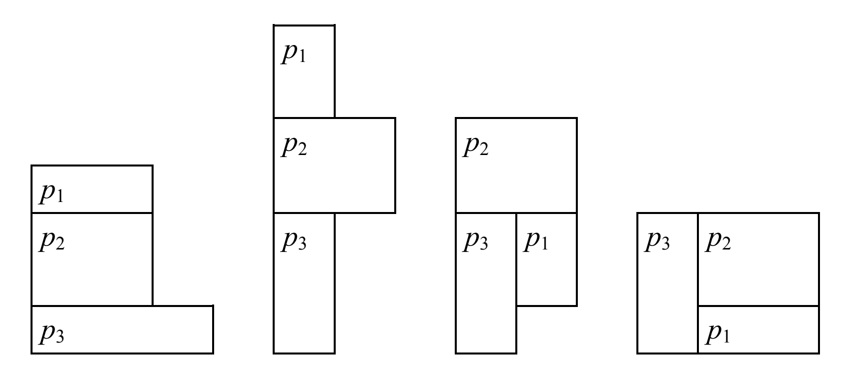
	Primera	Mejor	Todas
Núm. nodos generados	57	69	166
Núm. nodos analizados	48	58	159
Núm. nodos podados	0	10	0

Podemos observar que de nuevo la columna "Todas" consigue los mismos valores, y por la misma razón en este ejemplo, los valores de la columna "Mejor" no cambian. Sin embargo, vemos un empeoramiento notable de los valores en la columna "Primera". El motivo es obvio, pues al recorrer el árbol en anchura necesitamos generar muchos más nodos hasta llegar a la primera hoja solución.

7.6 LA COLOCACIÓN ÓPTIMA DE RECTÁNGULOS

Supongamos que disponemos de n piezas planas rectangulares p_1, p_2, ..., p_n, cada una con un área (a_i,b_i) $(1 \leq i \leq n)$, que precisamos encajar en un tablero plano rectangular T. El problema consiste en encontrar una disposición de las n piezas de forma que el tablero que necesitamos para contenerlas a todas sea de área mínima.

Por ejemplo, sean las piezas $p_1=(1,2)$, $p_2=(2,2)$ y $p_3=(1,3)$. El siguiente diagrama muestra cuatro disposiciones distintas de las tres piezas:



Como puede observarse, el área de los rectángulos que los recubren en cada uno de los casos es 12 (4x3), 14 (7x2), 10 (5x2) y 9 (3x3).

Solución (☞)

Este problema plantea dos dificultades principales. En primer lugar la de cómo generar el árbol de expansión pues, como veremos más adelante, las formas usuales de planteamiento de cualquier problema de Ramificación y Poda no valen para este caso.

La segunda dificultad es un problema de recursos, pues el árbol que se maneja es muy grande, y por tanto el número de nodos que se genera supera pronto la capacidad del ordenador. Incluso para el ejemplo del enunciado, con sólo tres piezas pequeñas, algunas estrategias agotan enseguida la memoria disponible.

Comenzaremos analizando la construcción del árbol de expansión para el problema. En primer lugar hemos de plantear la solución como una secuencia de decisiones, una en cada paso o etapa. La primera idea que intentamos llevar a cabo es la de ir colocando una pieza en cada paso. Así en la etapa k colocaremos la pieza p_k $(1 \leq k \leq n)$ adyacente a las que ya tenemos, y para cada una de ellas será suficiente almacenar la posición en donde la hemos colocado. Sin embargo, esta estrategia no es válida puesto que al ir colocando las piezas por orden y cada una junto a las que ya teníamos colocadas, no cubrimos todas las posibilidades. Por ejemplo, de esta forma no podríamos tener la pieza número uno junto a la tercera, y detrás de ésta la segunda. Por este motivo, en cada paso necesitamos explorar todas las piezas aún no colocadas, y no la pieza p_k en concreto.

Por otro lado, para cada una de estas piezas tenemos múltiples opciones, pues podemos colocarlas vertical u horizontalmente (a menos que la pieza sea simétrica) y alrededor del conjunto de piezas que ya tenemos situadas.

En cuanto a nuestra representación de la solución, la forma más sencilla es la de disponer de un tablero en donde marcar las casillas ocupadas. El tablero puede ser implementado mediante una matriz de número naturales, en donde el valor 0 indica una posición libre y un valor $k > 0$ indica que esa posición está ocupada por la pieza p_k.

Con esto en mente, ya podemos atacar la implementación del tipo abstracto de datos que representa los nodos.

```
CONST N = ...; (* numero de piezas *)
CONST LMAX = ...; (* longitud maxima de una pieza (alto o ancho) *)
CONST XMAX = N*LMAX; YMAX=N*LMAX; (* tam. maximo del tablero *)

TYPE tablero = ARRAY[0..XMAX],[0..YMAX] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
      t:tablero; (* tablero asociado al nodo *)
      k:CARDINAL;  (* nivel *)
      xmax,ymax:CARDINAL; (* area acumulada *)
      puestas:ARRAY [1..N] OF BOOLEAN;  (* piezas ya colocadas *)
    END;
```

Además del tablero con la solución construida hasta ese momento y el nivel, las otras componentes del registro indican el área del rectángulo que contiene a las piezas colocadas y un vector que indica qué piezas están ya situadas y cuáles quedan por colocar.

Necesitaremos además dos variables globales al módulo para almacenar la cota superior (el área) alcanzada por la mejor solución hasta el momento y el área de las piezas que debemos colocar:

```
VAR cota:CARDINAL;
VAR piezas:ARRAY[1..N] OF RECORD x,y:CARDINAL END;
```

Estas variables serán inicializadas en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
    cota:=MAX(CARDINAL);
    piezas[1].x:=1; piezas[1].y:=2;
    piezas[2].x:=2; piezas[2].y:=2;
    piezas[3].x:=1; piezas[3].y:=3;
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de contener el nodo raíz del árbol de expansión:

```
PROCEDURE NodoInicial():nodo;
  VAR n:nodo; i,j:CARDINAL;
BEGIN
  NEW(n); n^.k:=0; n^.xmax:=0; n^.ymax:=0; (* origen *)
  FOR i:=1 TO N DO n^.puestas[i]:=FALSE END;
  FOR i:=0 TO XMAX DO FOR j:=0 TO YMAX DO n^.t[i,j]:=0 END END;
  RETURN n;
END NodoInicial;
```

Como podemos observar, inicialmente el tablero se encuentra vacío. Por otro lado, la estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo va a generar un hijo por cada posición posible de cada una de las piezas aún no incluidas en el tablero. Este hecho es el que produce un árbol de expansión tan grande:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR  i,nhijos:CARDINAL; p:nodo;
        inicial,basura:BOOLEAN; a,b:CARDINAL;
BEGIN
 nhijos:=0;
 inicial:=(n^.xmax=0)AND(n^.ymax=0); (* esta vacio? *)
 FOR i:=1 TO N DO  (* generamos los hijos *)
   IF NOT n^.puestas[i] THEN
    FOR a:=0 TO n^.xmax+1 DO
     FOR b:=0 TO n^.ymax+1 DO
      Copiar(n,p);
      IF ColocarPieza(inicial,p,i,a,b,piezas[i].x,piezas[i].y) THEN
        INC(nhijos); INC(p^.k); hijos[nhijos-1]:=p;
      ELSE Eliminar(p);
      END;
      IF piezas[i].x<>piezas[i].y THEN (* no simetrica *)
        Copiar(n,p);
        IF ColocarPieza(inicial,p,i,a,b,piezas[i].y,piezas[i].x)THEN
          INC(nhijos); INC(p^.k); hijos[nhijos-1]:=p;
        ELSE Eliminar(p);
        END
      END
     END
    END
   END
 END;
 RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
   PROCEDURE Copiar(VAR n1,n2:nodo);
```

```
BEGIN
    NEW(n2);
    n2^.xmax:=n1^.xmax; n2^.ymax:=n1^.ymax;
    n2^.t:=n1^.t; n2^.puestas:=n1^.puestas;
    n2^.k:=n1^.k;
END Copiar;
```

Y también de otra función para determinar si una pieza puede ser colocada o no en una determinada posición:

```
PROCEDURE ColocarPieza (inicial:BOOLEAN; (* primera pieza?*)
                        VAR n:nodo;      (* nodo vivo *)
                        p:CARDINAL;      (* num. pieza a poner *)
                        x,y:CARDINAL;    (* donde ponerla *)
                        a,b:CARDINAL     (* largo y alto *)
                        ):BOOLEAN;       (* puedo ponerla? *)
   VAR i,j:CARDINAL; conexa:BOOLEAN;
BEGIN
   IF (inicial)AND((x<>0)OR(y<>0)) THEN RETURN FALSE END;
   (* primero miramos que cabe *)
   IF ((x+a-1)>XMAX)OR((y+b-1)>YMAX) THEN RETURN FALSE END;
   (* despues miramos que no pisa a ninguna pieza *)
   FOR i:=x TO x+a-1 DO FOR j:=y TO y+b-1 DO
      IF n^.t[i,j]<>0 THEN RETURN FALSE END
   END END;
   (* despues miramos que sea adyacente a otra *)
   IF NOT inicial THEN
      conexa:=FALSE;
      IF x=0 THEN i:=0 ELSE i:=x-1 END;
      WHILE (i<=Max2(x+a,XMAX)) AND (NOT conexa) DO
         IF (((y>0)AND(n^.t[i,y-1]<>0))OR
            ((y+b<=YMAX)AND(n^.t[i,y+b]<>0))) THEN conexa:=TRUE END;
         INC(i)
      END;
      IF y=0 THEN j:=0 ELSE j:=y-1 END;
      WHILE (j<=Max2(y+b,YMAX)) AND (NOT conexa) DO
         IF (((x>0)AND(n^.t[x-1,j]<>0))OR
            ((x+a<=XMAX)AND(n^.t[x+a,j]<>0))) THEN conexa:=TRUE END;
         INC(j)
      END;
      IF NOT conexa THEN RETURN FALSE END;
   END;

   (* ahora la ponemos en el tablero *)
   FOR i:=x TO x+a-1 DO FOR j:=y TO y+b-1 DO
      n^.t[i,j]:=p
```

```
   END END;
   n^.puestas[p]:=TRUE;
   (* y ajustamos los nuevos bordes del tablero *)
   n^.xmax:=Max2(x+a-1,n^.xmax);
   n^.ymax:=Max2(y+b-1,n^.ymax);
   RETURN TRUE;
END ColocarPieza;
```

La dificultad de este problema reside en las funciones *Expandir* y *ColocarPieza*. Como podemos ver, la primera de ellas genera los nodos hijos de un nodo dado, y recorre el tablero buscando posiciones en donde situar cada una de las piezas aún no colocadas. Por cada pieza puede realizar hasta dos veces esta tarea, según disponga la pieza vertical u horizontalmente. Por su parte, la segunda función es la que decide si una posición es válida o no para colocar una pieza. Entendemos por válida que quepa en el tablero, no "pise" a ninguna otra, y sea adyacente a alguna de las piezas previamente colocadas.

También es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuya área hasta el momento supere la alcanzada por una solución ya encontrada. Para eso definimos una función de coste para los nodos. Como buscamos la solución de menor área total, el valor que vamos a tomar es el del área acumulada hasta el momento:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
    RETURN Max2((n^.xmax+1)*(n^.ymax+1),AreaTotalPiezas);
END h;
```

donde *AreaTotalPiezas* es el área de todas la piezas, en este caso 9, que es el mejor de los casos posibles.

De las dos funciones siguientes, la primera calcula el valor asociado a una solución, y la segunda es la que va a permitir realizar la poda:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN (n^.xmax+1)*(n^.ymax+1);
END Valor;

PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de acomodar todas las piezas. Como en cada paso colocamos una, llegaremos a una hoja cuando el nivel del nodo sea *N*:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca, pues estamos suponiendo que el tablero es lo suficientemente grande para acomodar a todas la piezas.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de las soluciones.

Para los valores iniciales dados en el ejemplo, el algoritmo encuentra una disposición óptima de coste 9, que es una de las indicadas en el enunciado del problema. Los valores del árbol de expansión que se obtienen son:

Núm. nodos generados	1081
Núm. nodos analizados	80
Núm. nodos podados	982

Podemos plantearnos también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro modelo de programación, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola. Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente.

	LC	LIFO	FIFO
Núm. nodos generados	1081	681	1081
Núm. nodos analizados	80	59	80

Núm. nodos podados	982	596	982

Como puede observarse, para los valores del ejemplo trabaja mejor la estrategia LIFO, debido también al orden en el que se va generando el árbol de expansión. Éste es un buen ejemplo en donde la función de coste que hemos utilizado para implementar la estrategia LC no da buenos frutos.

También conviene destacar que éste es un ejemplo en donde la poda realiza una gran labor, pero se trata de la poda "a posteriori". Y hemos de señalar que aunque el número de nodos generados sea grande, el trabajo real del algoritmo, que viene dado por el número de nodos analizados, no es excesivo pese al gran número de nodos que se generan.

7.7 LA MOCHILA (0,1)

Recordemos el problema de la Mochila (0,1), enunciado por primera vez en el capítulo 4. Dados n elementos $e_1, e_2, ..., e_n$ con pesos $p_1, p_2, ..., p_n$ y beneficios $b_1, b_2, ..., b_n$, y dada una mochila capaz de albergar hasta un máximo de peso M (capacidad de la mochila), queremos encontrar cuáles de los n elementos hemos de introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima.

Esto es, hay que encontrar valores $(x_1, x_2, ..., x_n)$, donde cada x_i puede ser 0 ó 1, de forma que se maximice el beneficio, dado por la cantidad $\sum_{i=1}^{n} b_i x_i$, sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \le M$.

En este caso nos planteamos resolver el problema utilizando una estrategia LC.

Solución (☹)

Para construir el árbol de expansión del problema es necesario plantear la solución como una secuencia de decisiones, una en cada etapa o nivel del árbol. Para ello, vamos a representar la solución del problema mediante un vector, en donde en cada posición podrá encontrarse uno de los valores 1 ó 0, indicando si introducimos o no el elemento en cuestión.

Así, comenzando por el primer elemento, iremos recorriéndolos todos y decidiendo en cada paso si incluimos o no el elemento, por lo que cada nodo va a dar lugar a lo sumo a dos hijos. Sin pérdida de generalidad vamos a suponer los elementos ordenados de forma decreciente en cuanto a su ratio beneficio/peso para facilitar el cálculo de la función de coste, tal y como veremos más adelante.

Respecto a la poda, éste es un problema de maximización, mientras que el esquema visto en la introducción del capítulo (*RyP_lamejor()*) está diseñado para problemas de minimización. Pero el cambio es bien sencillo, pues basta con considerar la naturaleza dual de ambos problemas y utilizar el hecho de que para maximizar una función positiva v basta con minimizar la función $v' = -v$.

Comencemos entonces a definir la estructura de datos que contendrá a los nodos. En ellos se ha de almacenar información sobre la solución alcanzada hasta ese momento, y por tanto definimos:

```
CONST N = ...; (* numero de elementos distintos *);
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                         peso, (* peso acumulado *)
                         beneficio, (* beneficio acumulado *)
                         k:CARDINAL; (* nivel *)
                         s:solucion
                      END;
```

Necesitamos además tres variables globales al módulo "Nodos": una para almacenar la cota superior alcanzada hasta el momento, otra con la capacidad máxima de la mochila, y otra para guardar la tabla con los datos iniciales del problema. Obsérvese que esta tabla es global pues contiene la información sobre los propios elementos.

```
VAR cota:CARDINAL;
VAR capacidad:CARDINAL;
VAR tabla: ARRAY [1..N] OF RECORD beneficio,peso:CARDINAL END;
```

Estas variables serán inicializadas en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
   capacidad:=8;
   (* ordenados de forma decreciente por ratio beneficio/peso *)
   tabla[1].beneficio:=10;
   tabla[1].peso:=5;
   tabla[2].beneficio:=5;
   tabla[2].peso:=3;
   tabla[3].beneficio:=6;
   tabla[3].peso:=4;
   tabla[4].beneficio:=3;
   tabla[4].peso:=2;
END Nodos.
```

La función *NodoInicial* ha de generar un nodo vacío inicialmente:

```
PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i:CARDINAL;
BEGIN
    NEW(n);
    FOR i:=1 TO N DO n^.s[i]:=0 END;
```

```
    n^.peso:=0;
    n^.beneficio:=0;
    n^.k:=0;
    RETURN n;
END NodoInicial;
```

La estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar a lo sumo dos hijos, que corresponden a incluir el elemento o no en la mochila. Sólo serán generados aquellos nodos que sean válidos, esto es, si caben en la mochila, teniendo en cuenta la capacidad utilizada hasta el momento.

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,peso,plazo,beneficio,nhijos:CARDINAL; p:nodo;
BEGIN
    nhijos:=0;
    i:=n^.k+1;
    IF i>N THEN RETURN nhijos END; (* caso especial *)
    peso:=tabla[i].peso;
    beneficio:=tabla[i].beneficio;
    (* caso 0: no lo metemos *)
    INC(nhijos);
    Copiar(n,p);
    INC(p^.k); (* no se aumenta el peso ni el beneficio *)
    hijos[nhijos-1]:=p;
    (* caso 1: lo metemos *)
    IF n^.peso+peso<=capacidad THEN (* cabe! *)
        INC(nhijos);
        Copiar(n,p);
        p^.s[i]:=1;
        INC(p^.k);
        INC(p^.peso,peso);
        INC(p^.beneficio,beneficio);
        hijos[nhijos-1]:=p;
    END;
    RETURN nhijos;
END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
    VAR i:CARDINAL;
BEGIN
    NEW(n2);
    FOR i:=1 TO N DO n2^.s[i]:=n1^.s[i] END;
```

```
    n2^.peso:=n1^.peso;
    n2^.beneficio:=n1^.beneficio;
    n2^.k:=n1^.k;
END Copiar;
```

Por otro lado, es necesario implementar la función que realiza la poda. Comenzaremos primero definiendo la función de coste que vamos a asignar a cada nodo. Para ello vamos a considerar que los elementos iniciales están todos ordenados de forma decreciente por su ratio beneficio/peso. Según esto, cuando nos encontramos en el paso *k*-ésimo disponemos de un beneficio acumulado B_k. Por la forma en como hemos ido construyendo el vector, sabemos que:

$$B_k = \sum_{i=1}^{k} s[i] * tabla[i].beneficio\,.$$

Para calcular el valor máximo que podríamos alcanzar con ese nodo (B_M) procederemos de igual forma a como hicimos en la resolución de este problema utilizando la técnica de Vuelta Atrás (apartado 6.8). Así, vamos a suponer que rellenáramos el resto de la mochila con el mejor de los elementos que nos quedan por analizar. Como los tenemos dispuestos en orden decreciente de ratio beneficio/peso, éste mejor elemento será el siguiente (k+1). Este valor, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor al que podemos “aspirar” si seguimos por esa rama del árbol:

$$B_M = B_k + \left(capacidad - \sum_{i=1}^{k} s[i] * tabla[i].peso \right) \frac{tabla[k+1].beneficio}{tabla[k+1].peso}$$

Esto da lugar a la siguiente función de coste para un nodo dado:

```
PROCEDURE h(n:nodo):CARDINAL;
    VAR mejor:CARDINAL;
BEGIN
    IF EsSolucion(n) THEN RETURN n^.beneficio END;
    mejor:=CARDINAL((REAL(tabla[n^.k+1].beneficio)/
                     REAL(tabla[n^.k+1].peso))+0.5);
    RETURN n^.beneficio+(capacidad-n^.peso)*mejor
END h;
```

Obsérvese el carácter dual del problema de la mochila frente a los que hemos visto con anterioridad. Frente a un problema de minimización como teníamos en los anteriores, aquí nos planteamos la maximización del beneficio conseguido.

En general, los problemas de maximización de una función v se consiguen minimizando la función $-v$. Sin embargo, como es necesario trabajar con números positivos, utilizaremos el hecho de que dada una constante positiva t, el problema de minimizar una función f coincide con el de minimizar la función f+t. Uniendo ambas consideraciones, para maximizar nuestra función original v trataremos de

minimizar *MAX(CARDINAL)–v*, que es una función no negativa. Esto hace que definamos la función *Valor* como:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
    RETURN MAX(CARDINAL)-h(n);
END Valor;
```

De esta forma podremos podar, al igual que hacíamos en los otros problemas, aquellos nodos cuya penalización hasta el momento supere la alcanzada por una solución ya encontrada:

```
PROCEDURE EsAceptable(n:nodo):BOOLEAN;
BEGIN
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de tratar hasta el *N*-ésimo elemento:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca, pues siempre existe al menos una solución, que es la que representa el vector [0,0,...,0], es decir, siempre podemos no incluir ningún elemento. Ésta es, por ejemplo, la solución a un problema en donde los pesos de los elementos superen la capacidad de la mochila.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

7.8 LA MOCHILA (0,1) CON MÚLTIPLES ELEMENTOS

El problema de la Mochila (0,1) con múltiples elementos fue presentado en el apartado 5.17, y es una variación del problema de la Mochila (0,1) en donde en vez de tener *n* objetos distintos, de lo que disponemos es de *n* tipos de objetos. En definitiva, se trata de cambiar la restricción de que los números x_i sólo puedan tomar los valores 0 ó 1 por la de que sean enteros no negativos.

Nos piden dar una solución a este problema utilizando Ramificación y Poda, diseñando una función de coste adecuada.

Por otro lado, existe una variación del problema en donde se incorpora la restricción de que existe sólo un número limitado de objetos de cada tipo. Sería interesante modificar el algoritmo anterior para tener en cuenta esta restricción.

Solución (☺)

Este problema está muy ligado al anterior y va a presentar muy pocas diferencias frente a él. En primer lugar, la solución va a seguir estando representada por un vector, pero esta vez no será de ceros y unos, sino que podrá tomar valores enteros positivos. Y en segundo lugar, cada nodo no generará a lo sumo dos hijos, sino que podrá generar varios, tantos como le permita la capacidad de la mochila.

El primer cambio no se ve reflejado en el algoritmo desarrollado en el problema anterior, pues el tipo *nodo* ya permitía almacenar valores positivos mayores que uno. El segundo cambio tiene su reflejo en la función que expande los nodos:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,peso,plazo,beneficio,nhijos:CARDINAL;
    p:nodo;
BEGIN
    nhijos:=0;
    (* en cada etapa generamos los nodos hijos *)
    i:=n^.k+1;
    IF i>N THEN RETURN nhijos END; (* caso especial *)
    peso:=tabla[i].peso;
    beneficio:=tabla[i].beneficio;
    (* caso 0: no lo metemos *)
    INC(nhijos);
    Copiar(n,p);
    INC(p^.k); (* no se aumenta el peso ni el beneficio *)
    hijos[nhijos-1]:=p;


    (* resto de los casos: metemos 1, 2, ... unidades *)
    j:=1;
    WHILE n^.peso+(peso*j)<=capacidad DO (* caben j unidades *)
        INC(nhijos);
        Copiar(n,p);
        p^.s[i]:=j;
```

```
        INC(p^.k);
        INC(p^.peso,peso*j);
        INC(p^.beneficio,beneficio*j);
        hijos[nhijos-1]:=p;
        INC(j)
    END;
    RETURN nhijos;
END Expandir;
```

Las demás funciones del módulo "Nodos" quedan igual.

Respecto a la modificación de limitar el número de objetos de un tipo, en primer lugar necesitamos modificar la estructura de datos que almacena los datos globales sobre los elementos, para incluir la información sobre el número de objetos que disponemos de cada tipo:

```
VAR tabla:ARRAY[1..N]OF RECORD beneficio,peso,unidades:CARDINAL END;
```

y, por supuesto, incluir la inicialización de tales datos en el proceso de inicialización del módulo "Nodos":

```
tabla[1].unidades:=2; tabla[2].unidades:=2;
tabla[3].unidades:=2; tabla[4].unidades:=2;
```

Por otro lado, en la función *Expandir* hace falta tener en cuenta esta limitación:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,peso,plazo,beneficio,nhijos:CARDINAL; p:nodo;
BEGIN
    nhijos:=0;
    (* en cada etapa generamos los posibles nodos hijos *)
    i:=n^.k+1;
    IF i>N THEN RETURN nhijos END; (* caso especial *)
    peso:=tabla[i].peso;
    beneficio:=tabla[i].beneficio;
    (* caso 0: no lo metemos *)
    INC(nhijos);
    Copiar(n,p);
    INC(p^.k); (* no se aumenta el peso ni el beneficio *)
    hijos[nhijos-1]:=p;


    (* resto de los casos: metemos 1, 2, ... unidades *)
    j:=1;
    WHILE (n^.peso+(peso*j)<=capacidad)AND(j<=tabla[i].unidades) DO
        (* caben j unidades *)
        INC(nhijos);
        Copiar(n,p);
```

```
          p^.s[i]:=j;
          INC(p^.k);
          INC(p^.peso,peso*j);
          INC(p^.beneficio,beneficio*j);
          hijos[nhijos-1]:=p;
          INC(j)
      END;
      RETURN nhijos;
  END Expandir;
```

Si nos fijamos, la única diferencia entre esta función y la del apartado anterior es la condición del bucle que va generando los hijos. Ahora se pregunta no sólo si cabría un nuevo elemento de ese tipo, sino además si disponemos de él.

7.9 LA ASIGNACIÓN DE TAREAS

El problema de la asignación de tareas puede resolverse también utilizando una técnica de Ramificación y Poda. Recordemos que este problema consiste en, dadas *n* personas y *n* tareas, asignar a cada persona una tarea minimizando el coste de la asignación total, haciendo uso de una matriz de tarifas que determina el coste de asignar a cada persona una tarea.

Deseamos implementar dicho algoritmo utilizando la técnica de Ramificación y Poda y resolver el problema de minimizar el coste total para las dos siguientes matrices de tarifas, en donde las letras representan personas y los números tareas:

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

	1	2	3	4	5
a	11	17	8	16	20
b	9	7	12	6	15
c	13	16	15	12	16
d	21	24	17	28	26
e	14	10	12	11	15

Solución (☺)

En primer lugar hemos de construir el árbol de expansión del problema, y para ello es necesario plantear la solución como una secuencia de decisiones, una en cada etapa o nivel del árbol. Una forma fácil de realizar esto es considerando la estructura que va a tener la solución del problema.

En este caso la solución puede ser representada mediante un vector, cuyo *k*-ésimo elemento indica la tarea asignada a la persona *k*. Así, comenzando por la primera persona, en cada paso decidiremos qué tarea le asignamos de entre las que no hayan sido asignadas todavía, lo que implica que cada nodo generará a lo sumo *N–k* nodos hijos.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de

datos que representa los nodos, pues el resto del programa es fijo para este tipo de algoritmos.

Respecto a la información que debe contener cada uno de los nodos, hemos de conseguir que cada uno de ellos sea "autónomo", esto es, que contenga toda la información relevante para poder realizar los procesos de bifurcación, poda y reconstrucción de la solución encontrada hasta ese momento. En consecuencia, al menos ha de contar con el nivel en donde se encuentra y con el vector solución construido hasta ese instante. Por otro lado, también debe contener la información que permita realizar la poda. En este sentido vamos a incluir una matriz de tarifas modificada, en donde vamos a ir anulando las opciones que dejan de tener sentido en cada paso. Por ejemplo, si asignamos la tarea 3 a la persona a, ya no tiene sentido asignar la tarea 3 a nadie más.

Por otro lado necesitamos una función de coste LC para podar nodos. Por tratarse de un problema de minimización, dicha función va a representar una cota inferior (teórica, y por lo tanto no necesariamente alcanzable) de la solución del problema. Para ello, calcularemos los mínimos de los elementos de cada columna aún no asignados, puesto que éstas son las mejores tarifas que vamos a poder tener para cada tarea, independientemente de a quien se las asignemos. De hecho, ésta es una cota no necesariamente alcanzable, pues no estamos imponiendo la restricción de que no se puedan repetir trabajadores.

Con todo esto, comenzaremos definiendo el tipo *nodo* que utilizaremos en la implementación del algoritmo de Ramificación y Poda que resuelve el problema. Vamos a utilizar entonces la siguiente estructura de datos:

```
CONST N = ...; (* numero de personas y tareas *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE tarifas  = ARRAY[1..N],[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                        matriz:tarifas; (* matriz de tarifas *)
                        k:CARDINAL; (* nivel *)
                        s:solucion
                      END;
```

Además del vector solución y el nivel, la otra componente del registro es una matriz de tarifas, pero modificada para reflejar el hecho de que ya hay ciertas tareas asignadas. La forma de reflejar esta circunstancia es mediante la asignación de una valor ∞ a las tarifas de aquellas tareas que no puedan ser asignadas.

Necesitaremos además una variable *cota* global al módulo para almacenar la cota superior alcanzada por la mejor solución hasta el momento. Esta variable será inicializada en el cuerpo principal del módulo "Nodos":

```
BEGIN (* Nodos *)
   cota:=MAX(CARDINAL);
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de contener el nodo raíz del árbol de expansión:

```
PROCEDURE NodoInicial():nodo; (* para el ejemplo 1 *)
   VAR n:nodo; i,j:CARDINAL;
BEGIN
   NEW(n);
   FOR i:=1 TO N DO n^.s[i]:=0 END;
   n^.k:=0;
   n^.matriz[1,1]:=94;  n^.matriz[1,2]:=1;   n^.matriz [1,3]:=54;
   n^.matriz[1,4]:=68;  n^.matriz[2,1]:=74;  n^.matriz[2,2]:=10;
   n^.matriz[2,3]:=88;  n^.matriz [2,4]:=82; n^.matriz [3,1]:=62;
   n^.matriz [3,2]:=88; n^.matriz [3,3]:=8 ; n^.matriz [3,4]:=76;
   n^.matriz [4,1]:=11; n^.matriz [4,2]:=74; n^.matriz [4,3]:=81;
   n^.matriz [4,4]:=21;
   RETURN n;
END NodoInicial;
```

Como podemos observar, se asocia la matriz original al nodo origen. Por otro lado, la estrategia de ramificación está a cargo de la función *Expandir*. Cada nodo puede generar, como hemos dicho antes, a lo sumo *N–k* hijos, que son los correspondientes a los nodos aún no incluidos en el recorrido:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR nk,i,j,l,coste,nhijos:CARDINAL; p:nodo;
BEGIN
   nhijos:=0;
   nk:=n^.k+1;
   i:=n^.s[nk-1];
   IF nk>N THEN RETURN nhijos END; (* caso especial *)
   FOR j:=1 TO N DO
      IF NoEsta(n^.s,nk-1,j) THEN
         INC(nhijos);
         Copiar(n,p);
         p^.s[nk]:=j;
         Quitar(p^.matriz,nk,j);
         INC(p^.k);
         hijos[nhijos-1]:=p;
      END
   END;
   RETURN nhijos;
END Expandir;
```

Esta función hace uso de varios procedimientos que a continuación veremos. El primero de ellos permite duplicar un nodo:

```
PROCEDURE Copiar(VAR n1,n2:nodo);
   VAR i,j:CARDINAL;
BEGIN
```

```
    NEW(n2);
    n2^.s:=n1^.s; n2^.matriz:=n1^.matriz; n2^.k:=n1^.k;
END Copiar;
```

También utiliza otra función para determinar si una tarea está incluida o no en la solución:

```
PROCEDURE NoEsta(s:solucion;k,j:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO k DO
      IF s[i]=j THEN RETURN FALSE END
   END;
   RETURN TRUE;
END NoEsta;
```

Aparte de estas dos funciones, también necesita modificar la matriz de tarifas de un nodo, eliminando las opciones que ya no son válidas. Esto lo realiza mediante el siguiente procedimiento:

```
PROCEDURE Quitar(VAR m:tarifas;i,j:CARDINAL);
    VAR k,temp:CARDINAL;
BEGIN
    temp:=m[i,j]; (* lo guardamos para reponerlo despues *)
    FOR k:=1 TO N DO
        m[i,k]:=MAX(CARDINAL); m[k,j]:=MAX(CARDINAL);
    END;
    m[i,j]:=temp;
END Quitar;
```

Además es necesario implementar la función que realiza la poda. En este caso vamos a implementar una función que asigne un coste a un nodo:

```
PROCEDURE CosteCol(VAR m:tarifas;j:CARDINAL):CARDINAL;
   VAR i,c:CARDINAL;
BEGIN (* calcula el elemento minimo de una columna dada *)
   c:=m[1,j];
   FOR i:=2 TO N DO IF m[i,j]<c THEN c:=m[i,j] END END;
   RETURN c
END CosteCol;
PROCEDURE Coste(VAR m:tarifas):CARDINAL;
(* calcula la suma de los minimos de las columnas *)
   VAR i,j,coste:CARDINAL;
BEGIN
   coste:=0;
   FOR j:=1 TO N DO (* lo hacemos por columnas *)
```

```
      INC(coste,CosteCol(m,j));
    END;
    RETURN coste;
  END Coste;

  PROCEDURE h(n:nodo):CARDINAL;
  (* funcion de coste de la estrategia LC *)
  BEGIN
    RETURN Coste(n^.matriz);
  END h;
```

Y otra que permita podar aquellos nodos cuyo coste hasta el momento supere el alcanzado por una solución ya encontrada:

```
  PROCEDURE EsAceptable(n:nodo):BOOLEAN;
  BEGIN
    RETURN Valor(n)<=cota;
  END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
  PROCEDURE Valor(n:nodo):CARDINAL;
  BEGIN
    RETURN h(n);
  END Valor;
```

y que devuelve el coste acumulado hasta el momento. Esto tiene sentido ya que nos piden encontrar la solución de menor coste. Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos conseguido acomodar hasta la *N*-ésima tarea:

```
  PROCEDURE EsSolucion(n:nodo):BOOLEAN;
  BEGIN
    RETURN n^.k=N;
  END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca pues siempre existe al menos una solución, que es la que asigna una tarea a cada persona.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
  PROCEDURE Eliminar(VAR n:nodo);
```

```
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo “Nodos”. El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

Para los valores iniciales dados en el primer ejemplo, el algoritmo encuentra un asignación óptima de coste 97, que es la que representa el vector solución [4,2,3,1], esto es, asigna a la persona *a* la tarea 4, a la persona *b* la tarea 2, a la persona *c* la tarea 3 y a la persona *d* la tarea 1. En la exploración del árbol de expansión para estos datos obtenemos los siguientes valores:

Núm. nodos generados	38
Núm. nodos analizados	18
Núm. nodos podados	20

Obsérvese el buen funcionamiento de la estrategia LC, pues con sólo el análisis de 18 nodos consigue descubrir la asignación óptima.

Respecto al segundo ejemplo, el algoritmo encuentra un asignación óptima de coste 60, que es la representada por el vector [1,4,5,3,2], obteniéndose los siguientes valores de exploración del árbol de expansión:

Núm. nodos generados	167
Núm. nodos analizados	84
Núm. nodos podados	83

7.10 LAS *n* REINAS

El problema de las *n* reinas, ya expuesto en el apartado 6.2, consiste en encontrar una disposición de todas ellas en un tablero de ajedrez de tamaño *n*x*n* de forma que ninguna amenace a otra.

Necesitamos resolver este problema utilizando Ramificación y Poda mediante las estrategias FIFO y LIFO, y comparar ambas soluciones.

Solución (☺)

El estudio del árbol de expansión de este problema ya es conocido, y sólo recordaremos que se basa en construir un vector solución formado por *n* enteros positivos, donde el *k*-ésimo de ellos indica la columna en donde hay que colocar la reina de la fila *k* del tablero. En cada paso o etapa disponemos de *n* posibles opciones a priori (las *n* columnas), pero podemos eliminar aquellas columnas que

den lugar a un vector que no sea k-prometedor, esto es, que la nueva reina incorporada amenace a las ya colocadas.

Más formalmente, diremos que el vector s de n elementos es k-prometedor (con $1 \le k \le n$) si y sólo si para todo par de enteros i y j entre 1 y k se verifica que $s[i] \neq s[j]$ y $| s[i] - s[j] | \neq | i - j |$.

Esto da lugar a un árbol de expansión razonablemente manejable (del orden de 2000 nodos para $n = 8$) y por tanto convierte el problema en "tratable".

Veamos cómo la técnica de Ramificación y Poda aborda dos recorridos distintos de ese árbol, en profundidad y en anchura, y qué resultados obtiene.

Comenzaremos definiendo entonces el tipo abstracto de datos que representa los nodos. Como han de ser autónomos para poder abordar los procesos de ramificación, poda y reconstrucción de la solución con la información contenida en cada uno de ellos, la manera natural de implementarlos es como sigue:

```
CONST N = ...; (* dimension del tablero *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                           k:CARDINAL; s:solucion;
                        END;
```

En el registro, k indica el nivel y s contiene la solución construida hasta el momento. De esta forma, el nodo inicial que forma la raíz del árbol contendrá una solución vacía:

```
PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i,j:CARDINAL;
BEGIN
   NEW(n);
   FOR i:=1 TO N DO
      n^.s[i]:=0
   END;
   n^.k:=0;
   RETURN n;
END NodoInicial;
```

Respecto al proceso de ramificación, cada nodo puede generar hasta n nodos hijos, cada uno con la columna de la reina que ocupa la fila en curso. Lo que ocurre es que descartaremos todos aquellos que no den lugar a un vector solución k-prometedor:

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
    VAR i,j,nhijos:CARDINAL; p:nodo;
BEGIN
   nhijos:=0;
   i:=n^.k+1;
   IF i>N THEN RETURN nhijos END; (* caso especial *)
```

```
    FOR j:=1 TO N DO
      IF EsKprometedor(n^.s,i-1,j) THEN
        INC(nhijos);
        Copiar(n,p);
        p^.s[i]:=j;
        INC(p^.k);
        hijos[nhijos-1]:=p;
      END
    END;
    RETURN nhijos;
  END Expandir;
```

Las funciones auxiliares de las que hace uso la función *Expandir* son las siguientes:

```
  PROCEDURE Copiar(VAR n1,n2:nodo); (* duplica un nodo *)
  BEGIN
    NEW(n2);
    n2^.s:=n1^.s; n2^.k:=n1^.k;
  END Copiar;

  PROCEDURE EsKprometedor(s:solucion;k,j:CARDINAL):BOOLEAN;
    VAR i:CARDINAL;
  BEGIN
    FOR i:=1 TO k DO
      IF (s[i]=j)OR(ValAbs(s[i],j)=k+1-i) THEN RETURN FALSE END;
    END;
    RETURN TRUE;
  END EsKprometedor;
```

Esta función hace uso de la que calcula el valor absoluto de la diferencia de dos enteros no negativos:

```
  PROCEDURE ValAbs(a,b:CARDINAL):CARDINAL;
  (* valor absoluto de la diferencia de sus argumentos: |a-b| *)
  BEGIN
    IF a>b THEN RETURN a-b
    ELSE RETURN b-a
    END
  END ValAbs;
```

Otra función importante es aquella que determina cuándo un nodo es una hoja del árbol de expansión, esto es, una solución al problema. Para ello, basta ver que el vector solución construido es *n*-prometedor:

```
  PROCEDURE EsSolucion(n:nodo):BOOLEAN;
  BEGIN
```

```
   RETURN n^.k=N;
END EsSolucion;
```

Aparte de estas funciones, el resto de los procedimientos que se definen en el interfaz de este tipo abstracto de datos no presentan mayor dificultad.

En primer lugar, las funciones *EsAceptable*, *Valor*, *PonerCota* y *h* no intervienen en el desarrollo de este problema, pues no existen podas a posteriori, es decir, la poda de nodos se realiza durante el proceso de ramificación, y todos aquellos nodos que se generan son válidos porque o son solución del problema, o conducen a una de ellas. La función *Eliminar* es la que devuelve al sistema los recursos utilizados por un nodo:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

La función *NoHaySolucion* es necesaria en este caso porque hay tableros en donde este problema no tiene solución (p.e. para n=3).

```
PROCEDURE NoHaySolucion():nodo;
BEGIN
   RETURN NIL;
END NoHaySolucion;
```

Una vez implementado este módulo, las estrategias FIFO y LIFO que queramos analizar van a llevarse a cabo mediante el uso de una implementación adecuada del módulo "Estruc". Los resultados que hemos obtenido utilizando una y otra hasta conseguir encontrar la primera solución del problema son los siguientes:

	LIFO	FIFO
Núm. nodos generados	124	1965
Núm. nodos analizados	113	1665
Núm. nodos podados	0	0

Como puede apreciarse en la tabla, el recorrido en profundidad del árbol es el más adecuado, y consigue en este caso recorrer el mismo número de nodos que recorría el algoritmo de Vuelta Atrás hasta encontrar la primera solución. Aquí, la primera solución que se encuentra mediante el uso de la estrategia LIFO es la que representa el vector [8, 4, 1, 3, 6, 2, 7, 5].

Por otro lado, es normal que el recorrido en anchura tenga que analizar tantos nodos, pues hasta no llegar al último nivel del árbol no encuentra la solución. Obsérvese además cómo tiene que analizar todos los nodos hasta el nivel n–1 y

generar casi todos los nodos del nivel n antes de encontrar la primera solución, que en este caso es [1, 5, 8, 6, 3, 7, 2, 4].

También es fácil analizar cómo se comportan una y otra estrategia cuando lo que le pedimos es que calculen todas la soluciones y no se detengan al encontrar la primera. Esto se consigue sencillamente utilizando la función *RyP_todas()* del esquema que presentamos al principio del capítulo:

	LIFO	FIFO
Núm. nodos generados	2056	2056
Núm. nodos analizados	1965	1965
Núm. nodos podados	0	0

Para este caso ambas estrategias obtienen los mismos resultados antes de encontrar las 92 soluciones que posee el problema para $n = 8$, pues ambas han de recorrer todo el árbol.

Por último, hacer notar que en ningún caso se podan nodos pues, como hemos señalado anteriormente, la poda se realiza durante el proceso de expansión, y no a posteriori.

7.11 EL FONTANERO CON PENALIZACIONES

Supongamos que un fontanero tiene N avisos pendientes, y que cada uno de ellos lleva asociado una duración (los días que tarda en realizarse), un plazo límite, y una penalización en caso de que no se ejecute dentro del plazo límite establecido para él (lo que deja de ganar). Por ejemplo, para el caso de cuatro avisos ($N = 4$) podemos tener los siguientes datos:

	1	2	3	4
Duración	2	1	2	3
Plazo límite	3	4	4	3
Penalización	5	15	13	10

En dicha tabla la duración y los plazos están expresados en días, y la penalización en miles de pesetas. Se pide determinar la fecha de comienzo de cada una de las tareas (cero si se decide no realizarla) de forma que la penalización total sea mínima.

Solución (☺)

Comenzaremos analizando la construcción del árbol de expansión para el problema. En primer lugar, hemos de plantear la solución como una secuencia de decisiones, una en cada paso o etapa.

Para ello, nuestra solución estará formada por un vector, con un elemento para cada una de las tareas. Cada uno de estos elementos contendrá el día de comienzo de la tarea correspondiente. Utilizaremos el valor 0 para indicar que tal tarea no se realiza.

De esta forma, inicialmente el vector estará vacío, y en el paso k-ésimo tomaremos la decisión de si hacemos o no la tarea número k, y en caso de decidir hacerla, cuál será su día de comienzo. Por tanto, los valores que puede en principio tomar el elemento en posición k del vector ($1 \leq k \leq N$) estarán comprendidos entre 0 y ($p - d + 1$), siendo p el plazo y d la duración de tal tarea. Sin embargo, todos esos valores no tienen por qué ser válidos; al incluir una tarea habrá de comprobarse que no se solape con las tareas que ya tenía asignadas el vector. Este mecanismo es el que va construyendo el árbol de expansión para este problema.

Teniendo en cuenta las consideraciones realizadas en la introducción de este capítulo, será suficiente realizar el módulo que implementa el tipo abstracto de datos que representa los nodos, pues el resto del programa es fijo para este tipo de algoritmos.

Para ello, comenzaremos definiendo el tipo *nodo* que representará el vector que hemos mencionado anteriormente. Utilizaremos entonces la siguiente estructura de datos:

```
CONST N = ...; (* numero de tareas *)
TYPE solucion = ARRAY[1..N] OF CARDINAL;
TYPE nodo = POINTER TO RECORD
                          penalizacion,k:CARDINAL; s:solucion
                      END;
```

Además del vector solución, las otras dos componentes del registro indican la penalización acumulada hasta el momento y la etapa en curso (k). De esta forma conseguimos que cada nodo sea "autónomo", esto es, que cada uno contenga toda la información relevante para poder realizar los procesos de bifurcación, poda y reconstrucción de la solución encontrada hasta ese momento.

Necesitaremos además dos variables globales al módulo. Una para almacenar la cota superior alcanzada por la mejor solución hasta el momento y otra para guardar la tabla con los datos iniciales del problema:

```
VAR cota:CARDINAL;
VAR tabla:ARRAY [1..N] OF RECORD
                           duracion,plazo,penalizacion:CARDINAL
                          END;
```

Estas variables serán inicializadas en el cuerpo principal del módulo "Nodos":

```
   BEGIN (* Nodos *)
      cota:=MAX(CARDINAL);
```

```
   tabla[1].duracion:=2; tabla[1].plazo:=3;
   tabla[1].penalizacion:=5;
   tabla[2].duracion:=1; tabla[2].plazo:=4;
   tabla[2].penalizacion:=15;
   tabla[3].duracion:=2; tabla[3].plazo:=4;
   tabla[3].penalizacion:=13;
   tabla[4].duracion:=3; tabla[4].plazo:=3;
   tabla[4].penalizacion:=10;
END Nodos.
```

Veamos ahora las funciones de este módulo. En primer lugar la función *NodoInicial* habrá de generar un nodo vacío:

```
PROCEDURE NodoInicial():nodo;
   VAR n:nodo; i:CARDINAL;
BEGIN
   NEW(n);
   FOR i:=1 TO N DO n^.s[i]:=0 END;
   n^.penalizacion:=0; n^.k:=0;
   RETURN n;
END NodoInicial;
```

La estrategia de ramificación está a cargo de la función "*Expandir*". Cada nodo puede generar, como hemos dicho antes, a lo sumo ($p - d + 2$) hijos, que son los correspondientes a no realizar la tarea o realizarla comenzando en los días 1, 2, ..., ($p - d + 1$). Esta función sólo generará aquellos nodos que sean válidos, esto es, que sean compatibles con las tareas asignadas previamente.

```
PROCEDURE Expandir(n:nodo;VAR hijos:ARRAY OF nodo):CARDINAL;
   VAR i,j,penalizacion,plazo,duracion,nhijos:CARDINAL; p:nodo;
BEGIN
   nhijos:=0;
   (* en cada etapa generamos los valores de la siguiente tarea *)
   i:=n^.k+1;
   IF i>N THEN RETURN nhijos END; (* caso especial *)
   penalizacion:=tabla[i].penalizacion;
   plazo:=tabla[i].plazo;
   duracion:=tabla[i].duracion;
   (* caso 0: no hacemos esa tarea *)
   INC(nhijos);
   Copiar(n,p);
   INC(p^.k);
   INC(p^.penalizacion,penalizacion);
   hijos[nhijos-1]:=p;
   (* resto de los casos *)
   FOR j:=1 TO (plazo-duracion+1) DO
```

```
    (* comprobamos que es compatible con el resto de tareas *)
      IF EsCompatible(n,i,j,duracion) THEN
        INC(nhijos);
        Copiar(n,p);
        p^.s[i]:=j; INC(p^.k);(* aqui no hay penalizacion *)
        hijos[nhijos-1]:=p;
      END;
    END;
    RETURN nhijos;
  END Expandir;
```

Esta función hace uso de un procedimiento que permite duplicar un nodo:

```
  PROCEDURE Copiar(VAR n1,n2:nodo);
    VAR i,j:CARDINAL;
  BEGIN
    NEW(n2);
    FOR i:=1 TO N DO n2^.s[i]:=n1^.s[i] END;
    n2^.penalizacion:=n1^.penalizacion;
    n2^.k:=n1^.k;
  END Copiar;
```

Y también de una función que decide si la decisión a tomar es compatible con las asignaciones previamente almacenadas en el vector:

```
  PROCEDURE EsCompatible(n:nodo;nivel,comienzo,duracion:CARDINAL)
                                                          :BOOLEAN;
    VAR i,fin,com:CARDINAL;
  BEGIN
    FOR i:=1 TO nivel-1 DO
      com:=n^.s[i];
      fin:=com+tabla[i].duracion-1;
      IF com<>0 THEN
       IF NOT((com>(comienzo+duracion-1))OR(fin<comienzo)) THEN
         RETURN FALSE
        END;
      END;
    END;
    RETURN TRUE;
  END EsCompatible;
```

Por otro lado, también es necesario implementar la función que realiza la poda. En este caso, vamos a podar aquellos nodos cuya penalización hasta el momento supere la alcanzada por una solución ya encontrada:

```
  PROCEDURE EsAceptable(n:nodo):BOOLEAN;
  BEGIN
```

```
   RETURN Valor(n)<=cota;
END EsAceptable;
```

Esta función hace uso de otra que es necesario implementar:

```
PROCEDURE Valor(n:nodo):CARDINAL;
BEGIN
   RETURN n^.penalizacion;
END Valor;
```

que devuelve la penalización de la solución construida hasta el momento. Esto tiene sentido pues nos piden encontrar la solución de menor penalización.

Veamos ahora la función de coste para los nodos. Como buscamos la solución de menor penalización, este valor se presenta como un buen candidato para tal función:

```
PROCEDURE h(n:nodo):CARDINAL;
BEGIN
   RETURN n^.penalizacion;
END h;
```

Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución. En nuestro caso consiste en decidir cuándo hemos sido capaces de acomodar hasta la tarea *N*-ésima:

```
PROCEDURE EsSolucion(n:nodo):BOOLEAN;
BEGIN
   RETURN n^.k=N;
END EsSolucion;
```

En cuanto a la función *NoHaySolucion*, que devuelve un valor especial para indicar que el problema no admite solución, sabemos que para este problema eso no ocurrirá nunca, pues siempre existe al menos una solución, que es la que representa el vector [0, 0, ..., 0], es decir, siempre podemos no hacer ninguna tarea, cuya penalización coincide con la suma de las penalizaciones de todas las tareas. Esta sería, por ejemplo, la solución al problema siguiente:

	1	2	3	4
Duración	4	5	6	7
Plazo límite	3	4	4	3
Penalización	5	15	13	10

en donde ninguna tarea puede realizarse por ser sus duraciones mayores a sus plazos límite.

Por su parte, la función *Eliminar* es la que va a devolver al sistema los recursos ocupados por un nodo, y es la que actúa como "destructor" del tipo abstracto de datos:

```
PROCEDURE Eliminar(VAR n:nodo);
BEGIN
   DISPOSE(n);
END Eliminar;
```

Con esto finaliza nuestra implementación del módulo "Nodos". El problema queda resuelto escogiendo la función del esquema que encuentra la mejor de todas las soluciones.

Para los valores iniciales dados en el enunciado, el algoritmo encuentra seis soluciones óptimas de penalización 15, que son:

[0, 1, 2, 0]
[0, 1, 3, 0]
[0, 2, 3, 0]
[0, 3, 1, 0]
[0, 4, 1, 0]
[0, 4, 2, 0]

Obteniéndose los siguientes valores de exploración del árbol de expansión:

Núm. nodos generados	51
Núm. nodos analizados	30
Núm. nodos podados	16

Nos podemos plantear también lo que ocurriría si hubiésemos escogido una estrategia distinta de la LC, esto es, LIFO o FIFO. Siguiendo nuestro esquema, bastaría con sustituir el módulo de implementación del tipo abstracto de datos "Estruc" acomodándolo a una pila o a una cola.

Estos cambios permitirán recorrer el árbol de expansión en profundidad o en anchura, respectivamente. Los valores que se obtienen para las tres estrategias son los siguientes:

	LC	LIFO	FIFO
Núm. nodos generados	51	48	58
Núm. nodos analizados	30	27	36
Núm. nodos podados	16	12	11

Como era de esperar, el peor caso es para la búsqueda en anchura. Además, para estos datos vemos cómo la estrategia LIFO mejora sensiblemente la LC; el hecho

de que sea un poco mejor depende sólo de los datos del problema. Para otros ejemplos los valores que se obtienen siguiendo esta estrategia son peores.

Por último, cabe preguntarse qué ocurre si deseamos buscar no la mejor, sino todas las posibles soluciones de este problema. Para este ejemplo los valores que se obtienen son:

Núm. nodos generados	58
Núm. nodos analizados	36
Núm. nodos podados	0

Por supuesto, estos valores se obtienen independientemente de la estrategia seguida (FIFO, LIFO o LC). Además, es curioso observar cómo los dos primeros valores coinciden con los obtenidos para la estrategia FIFO en la búsqueda de la mejor solución. La razón es bien sencilla, pues si vamos recorriendo el árbol de expansión en anchura necesitaremos recorrerlo entero para dar con la mejor solución, ya que todas las soluciones se encuentran siempre en el nivel *N*, y para llegar a él esta estrategia necesita haber construido completamente todos los niveles anteriores.